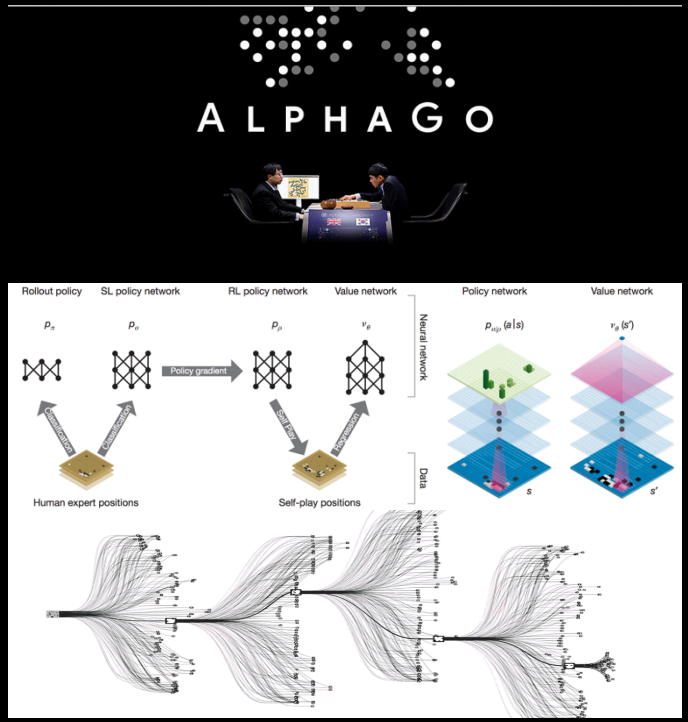
GPU Programming

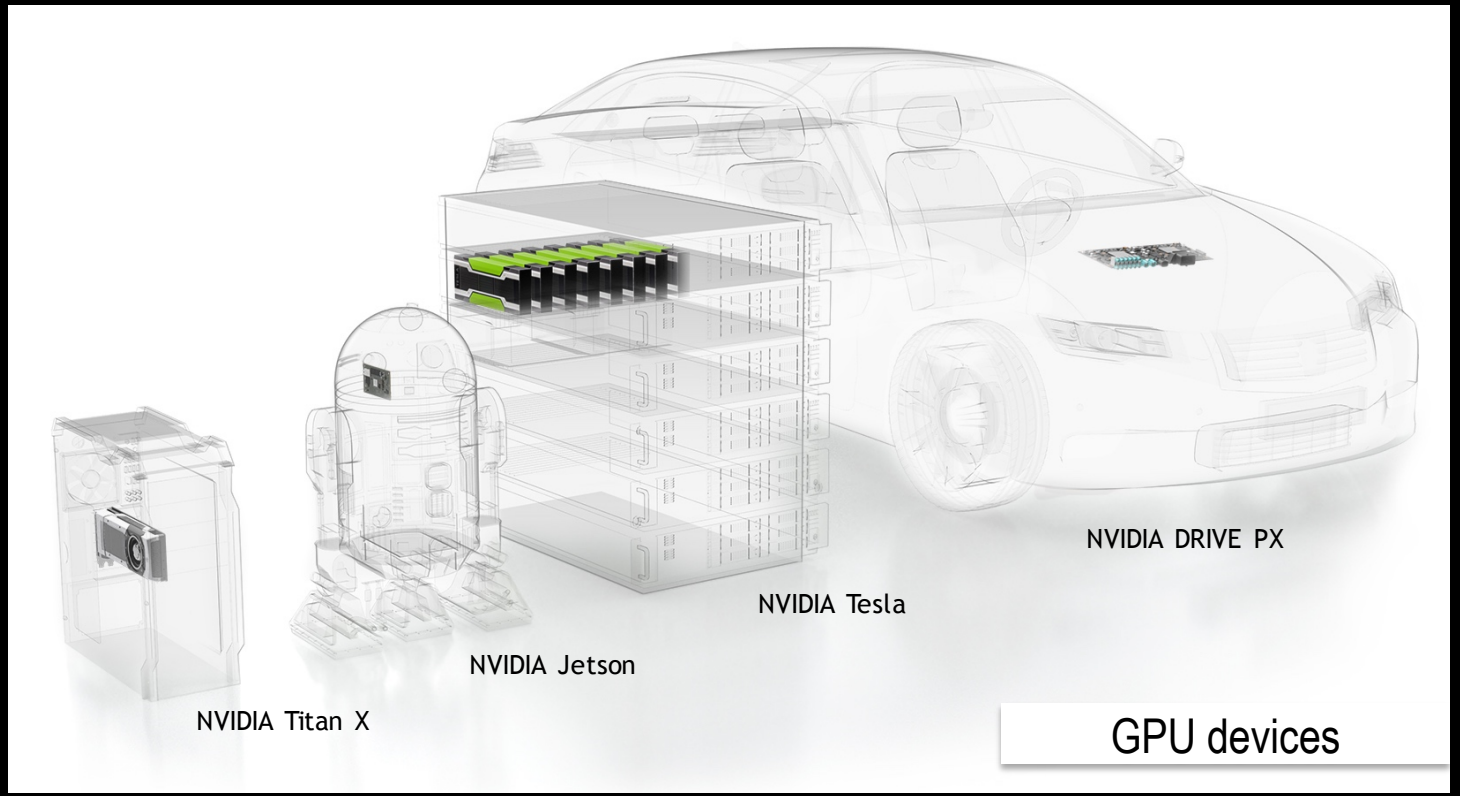
邓仰东
清华大学软件学院

AlphaGo

- Training DNNs: 3 weeks, 340 million training steps on 50 GPUs
- Play: Asynchronous multi-threaded search
  - Simulations on CPUs, policy and value DNNs in parallel on GPUs
  - Single machine: 40 search threads, 48 CPUs, and 8 GPUs
  - Distributed version: 40 search threads, 1202 CPUs and 176 GPUs
- Outcome: Beat both European and World Go champions in best of 5 matches



GPU for Deep Learning Everywhere



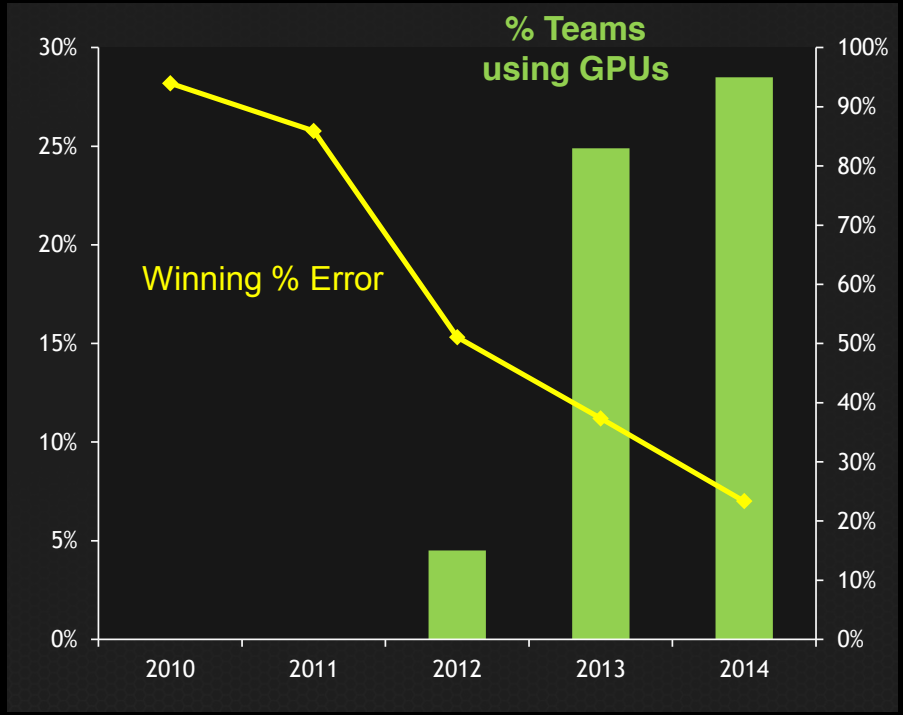
GPUs for Deep Learning

Image Recognition CHALLENGE

1.2M training images • 1000 object categories

Hosted by

IMAGENET

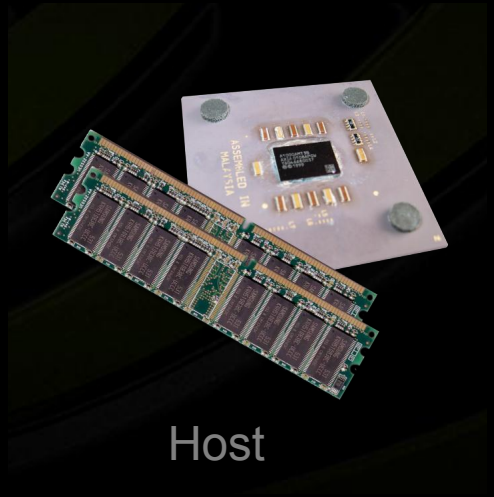


The Use of GPUs

- GPU accelerated applications and libraries
  - Incredibly useful for research, easy to use
  - Won't teach you much about GPUs
- Explicit GPU programming
  - Will teach you about GPUs
  - Some depth in one language will help you to understand libraries, applications, directives, and other GPU languages

Terminology

- Host The CPU and its memory (host memory)
- Device The GPU and its memory (device memory)



Host

Device

提纲

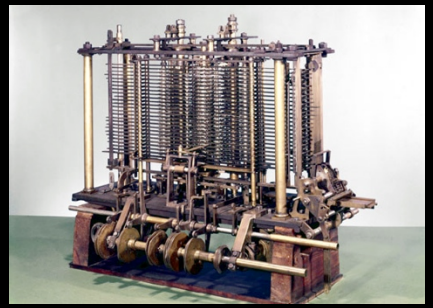
1. Road to GPU
2. CUDA Programming
3. Code Optimization
4. GPU Machine Learning

Parallel World

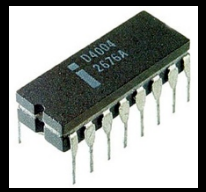
The Evolution of Computing Devices

Abacus

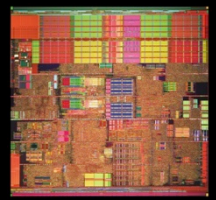
(BC600)



Analytical Engine (1871)

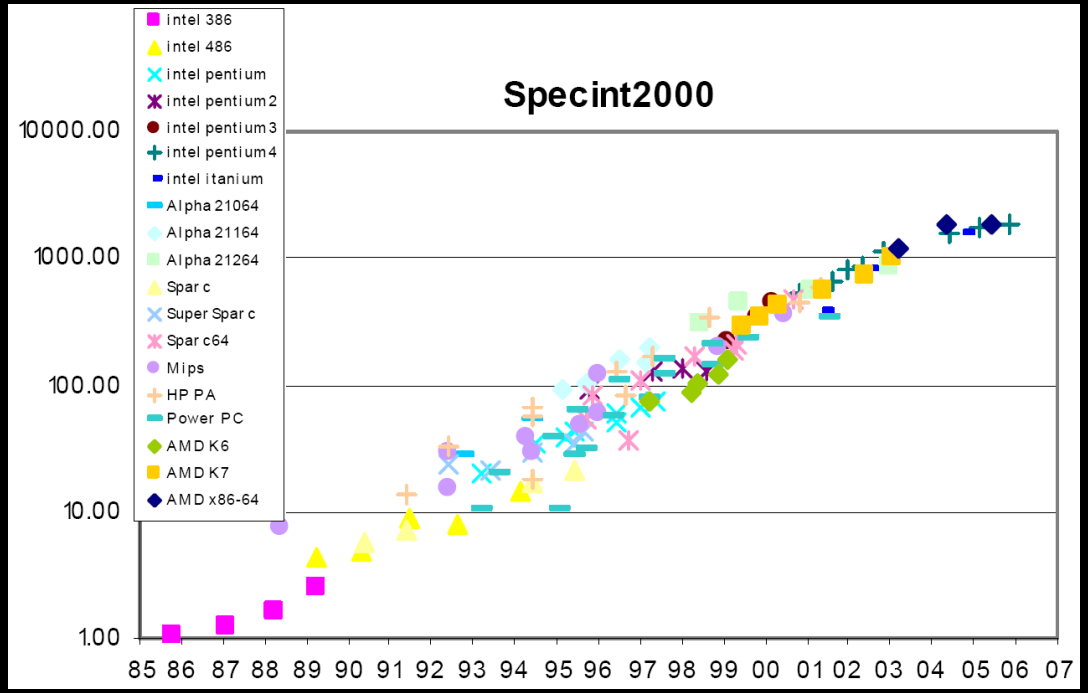


Single Core (1971)



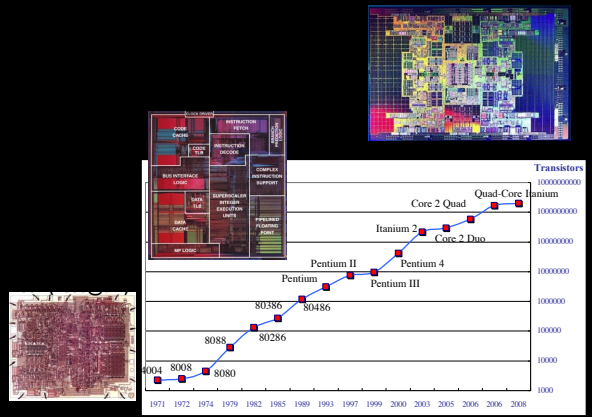
Pentium 4E or Prescott (2004)

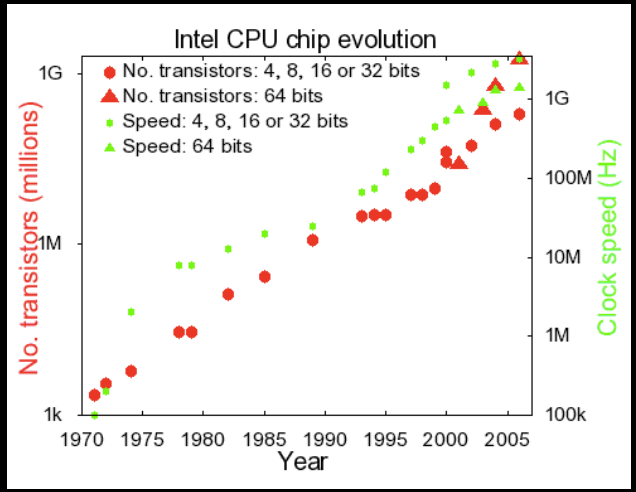
Single CPU Performance



How We Increased Single-Core Performance?

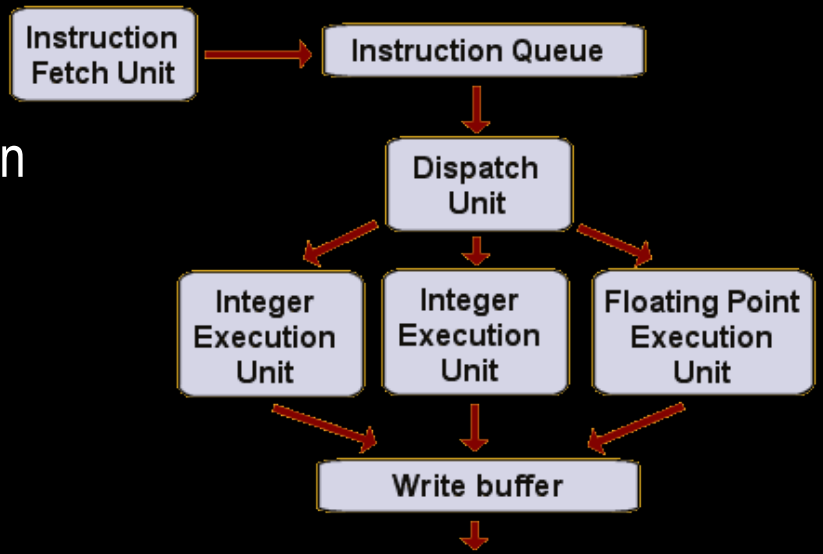
- Packing more transistors on a single chip
  - Enabled by Moore's Law
  - Representative
    - Superscalar, VLIW, SIMD, …
- Increasing clock frequency
  - Enabled by device scaling & pipeline





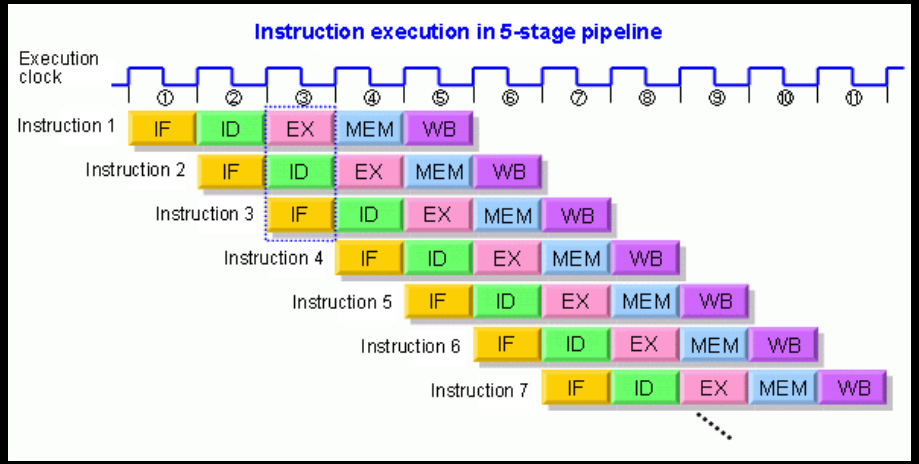
1980: Superscalar

- Deploy multiple functional units
- 10 CPI → 5 CPI
  - Implicit instruction level parallelism in function domain
    - 2-way to 8-way instruction issue
    - Out-of-order execution
    - Branch prediction
- Reached a limit when 1000s instructions on the fly



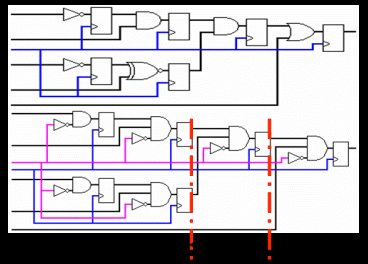
1990: Pipeline

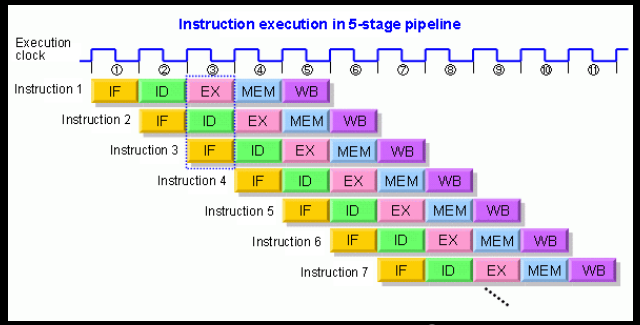
- 5 CPI → 0.5 CPI
  - Implicit instruction level parallelism in time domain
- Allows faster clock frequency



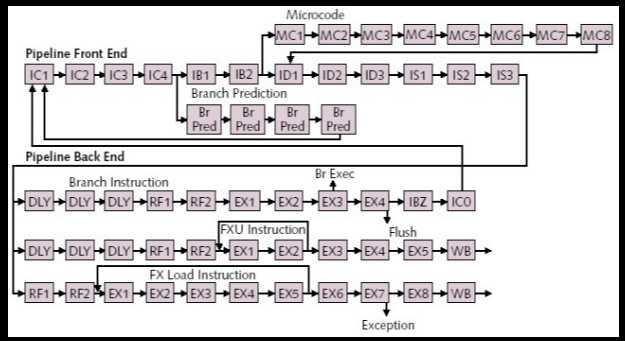
Faster Clock : Diminishing Return

- The increasing clock frequency
  - 30% from scaling
  - 70% from shortening of pipeline stage
    - Approaching limits (6-7 gates/stage)





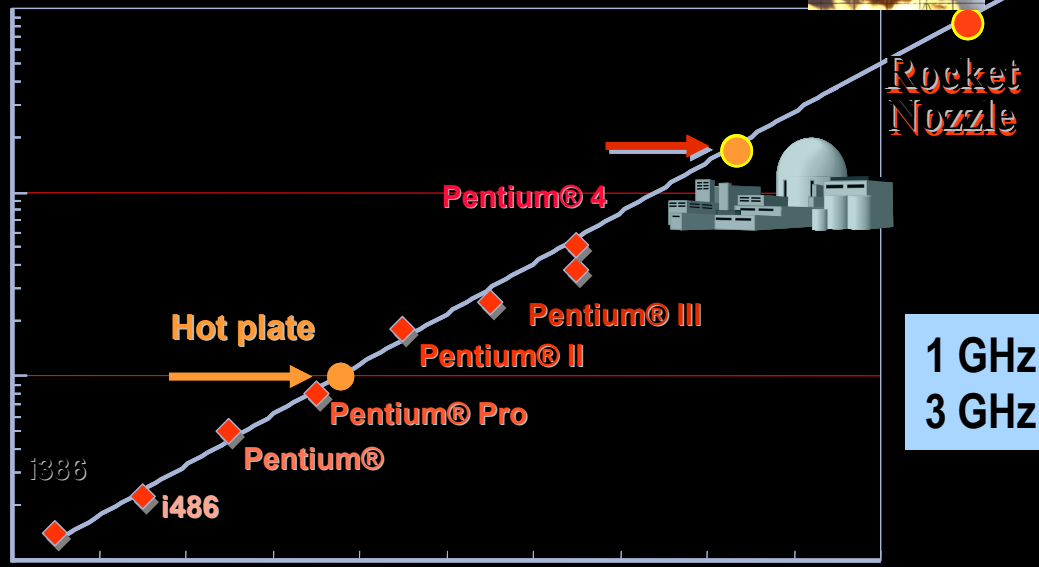
MIPS Pipeline



IBM Power7 Pipeline

"Power Wall"

$P = CV^2f$



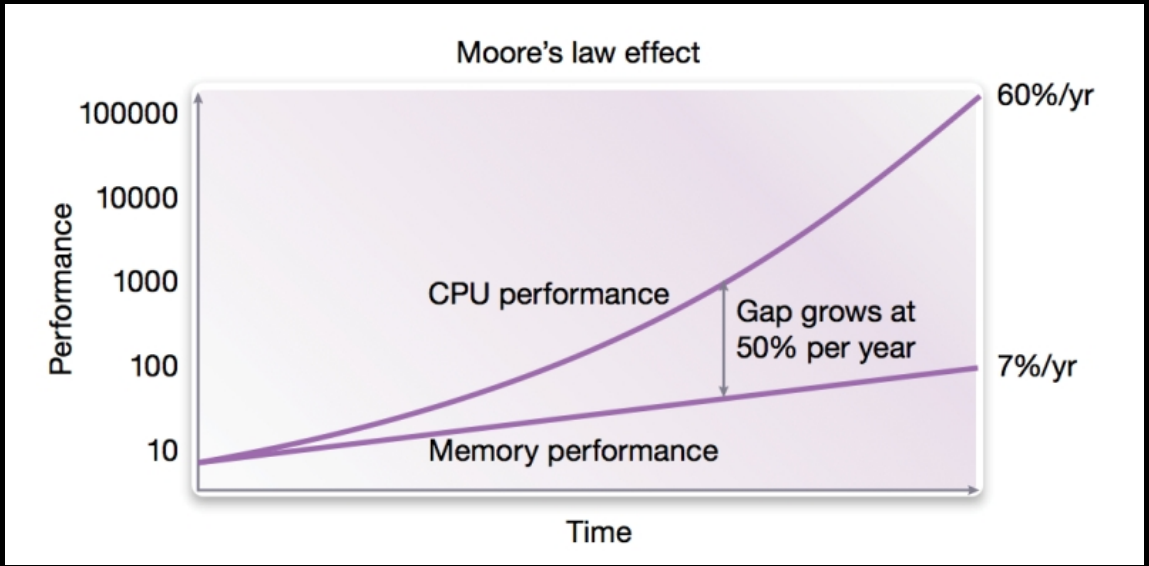
Sun's Surface

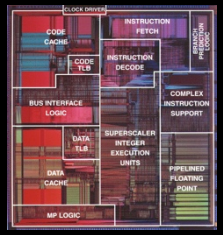
Rocket Nozzle

1 GHz CPU: ~25 W
3 GHz CPU: ~100 W

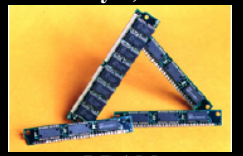
“Memory Wall”

- Extremely hard to reduce memory latency



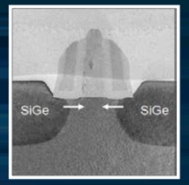


Logic
60%/yr (2X/1.5 yrs)

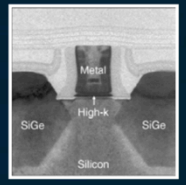


DRAM
9%/yr (2X/10 yrs)

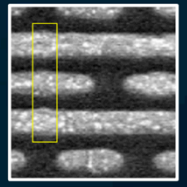
INTEL INNOVATION LEADERSHIP



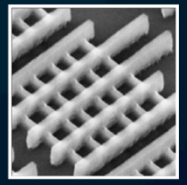
Strained Silicon



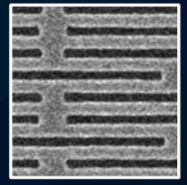
High-k Metal Gate



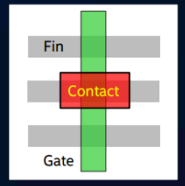
Self Align Via



FinFET Transistor



Hyper Scaling



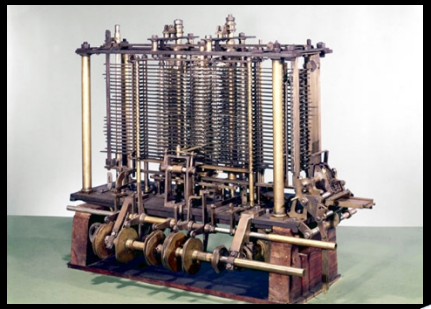
Hyper Scaling

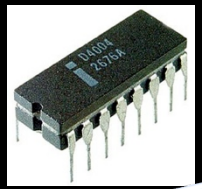
Intel	90nm		65nm		45nm		32nm		22nm		14nm		10nm					
Year	'03	'04	'05	'06	'07	'08	'09	'10	'11	'12	'13	'14	'15	'16	'17	'18	'19	'20
Others		90nm			65nm		40nm		28nm		20nm		16nm		10nm			
					Strained Silicon				High-k Metal Gate		Self Align Via		FinFET Transistor		Hyper Scaling		Hyper Scaling	
					>3 years later				>3 years later		>3 years later		>3 years later		~3 years later		??	

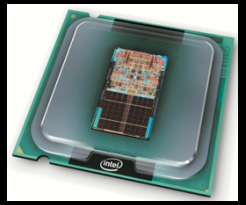
Intel developed all the major logic process innovations used by our industry over the past 15 years

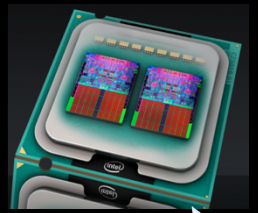
Source: Intel. 10 nm is based upon current expectations and available information.

The Evolution to Parallel Processors





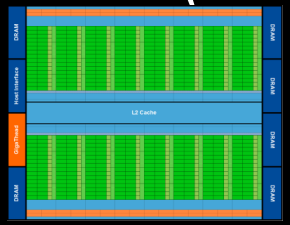




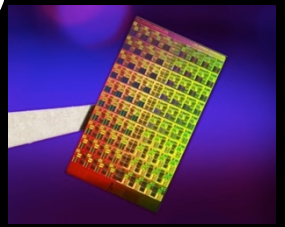
Analytical Engine (1871)

Single Core (1971)

Multi/Many Core (2006)



NVIDIA G80

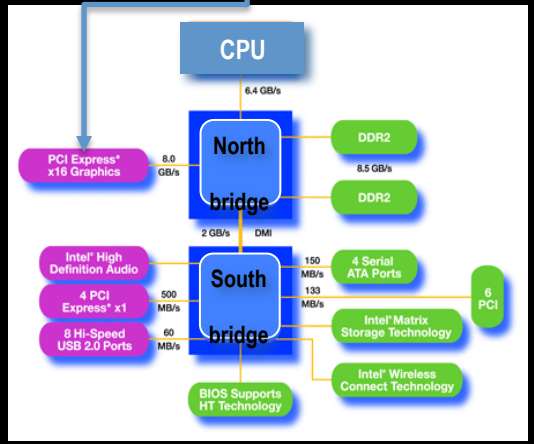


Intel MIC

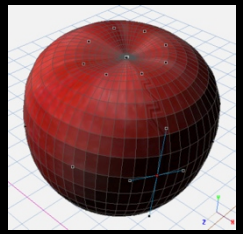

变形金刚5
最后的骑士
6月23日 变形出发
GTX 1080

Graphic Processing Unit (GPU)

- GPU – graphics processing unit
- Originally designed as a graphics processor
- Nvidia's GeForce 256 (1999) – first GPU
- Functionality
  - Single-chip processor for mathematically-intensive tasks
  - Transforms of vertices and polygons
  - Lighting
  - Polygon clipping
  - Texture mapping
  - Polygon rendering

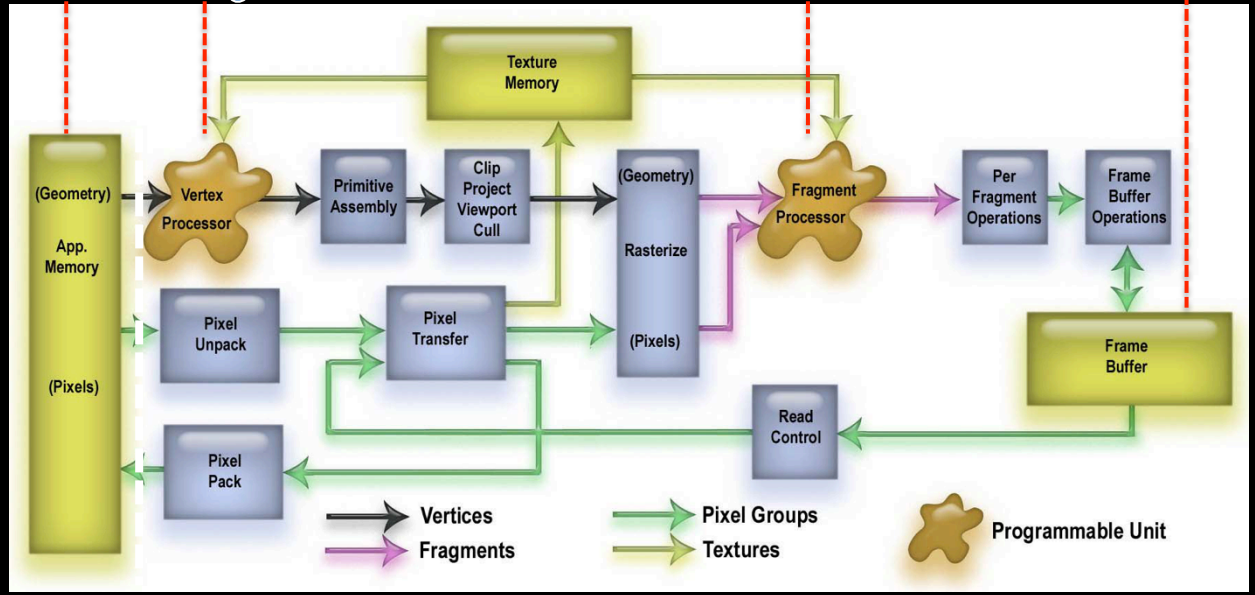


Graphics Pipeline

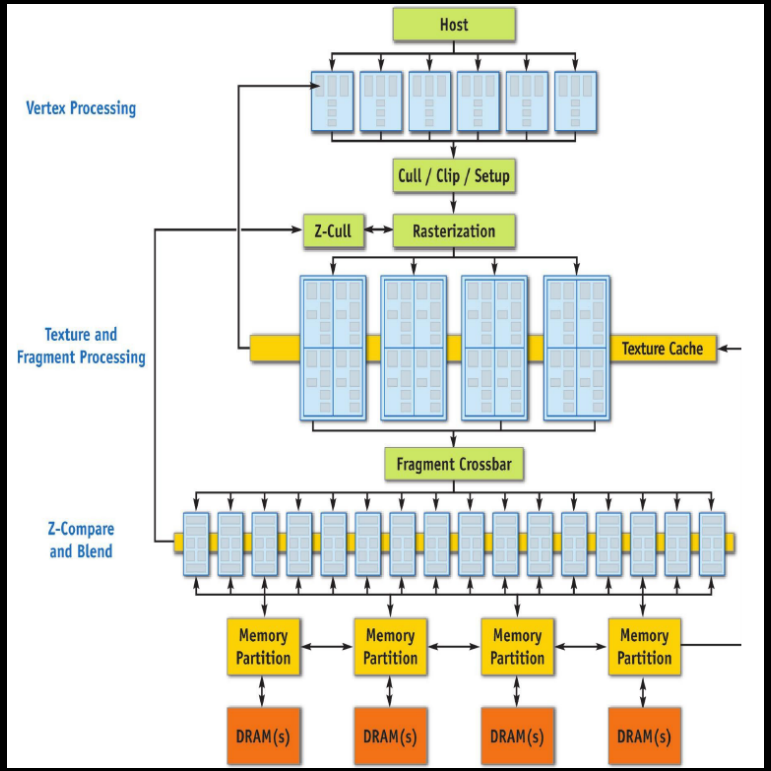


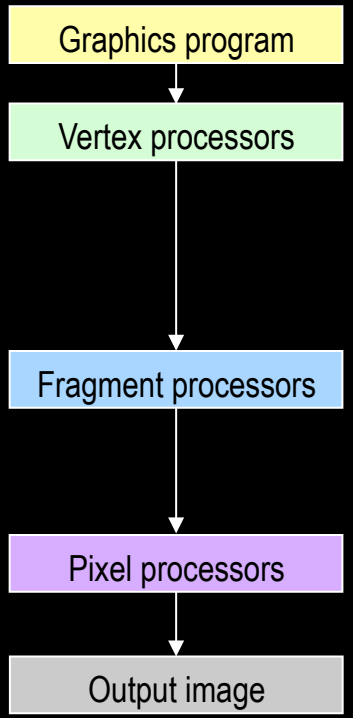
CPU

GPU



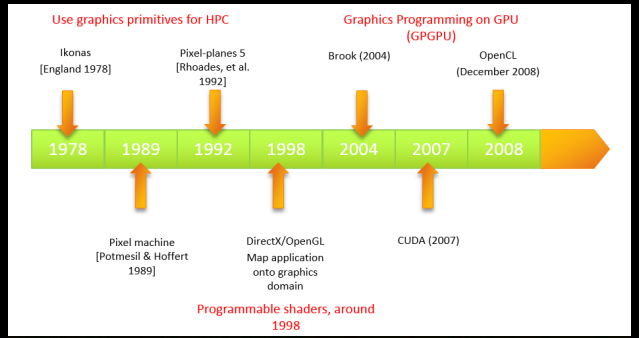
Traditional GPU Architecture

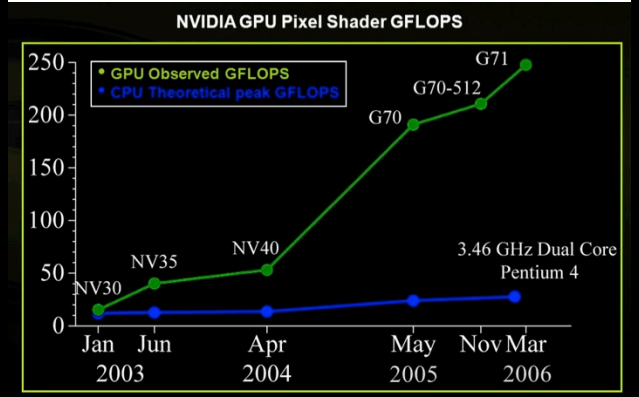




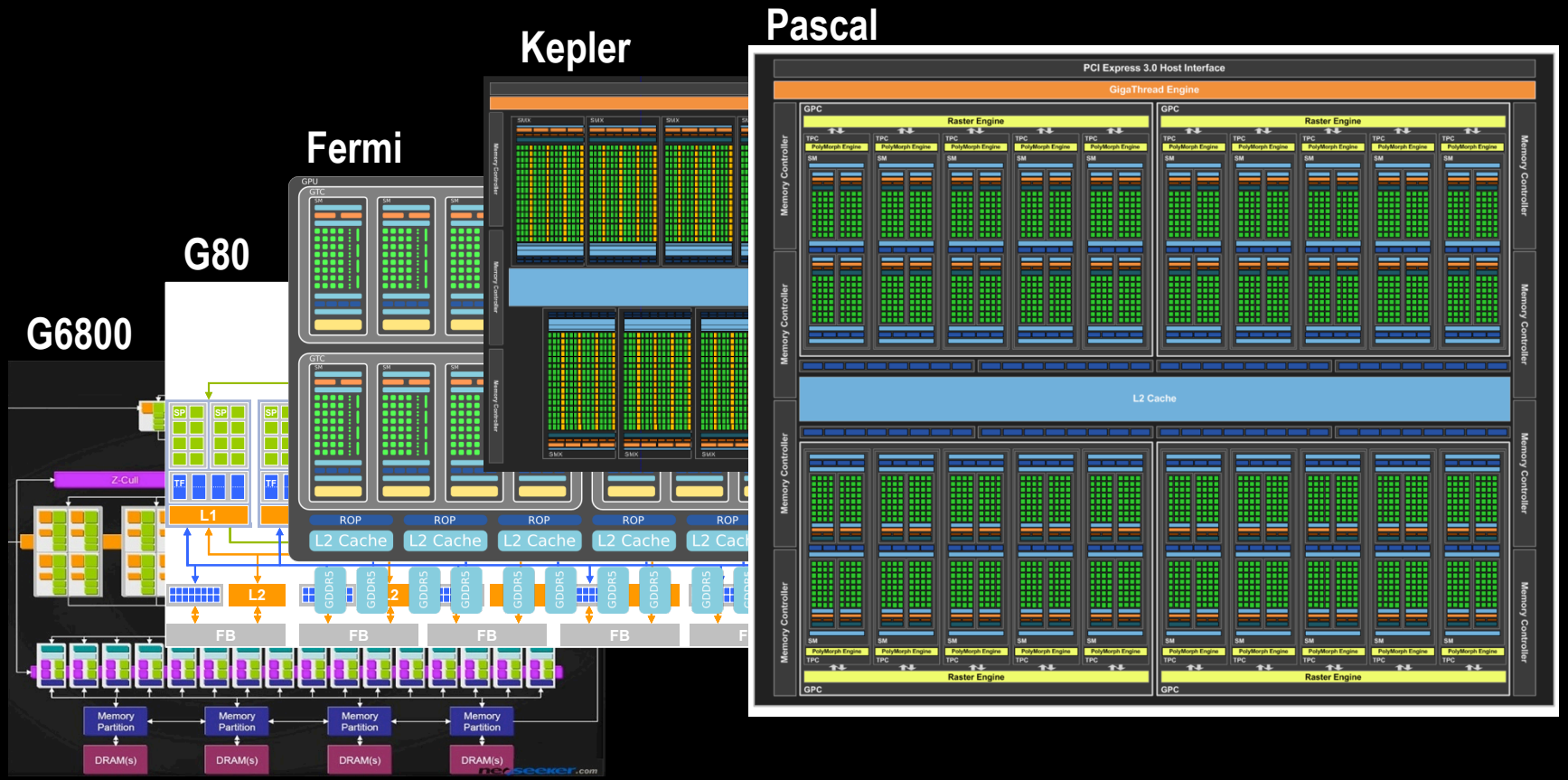
GPGPU

- General Purpose GPU Computing
  - 1999-2000 computer scientists from various fields started using GPUs to accelerate a range of scientific applications
  - GPU programming required the use of graphics APIs such as OpenGL and Cg
  - Since 2006, NVIDIA greatly invested in GPGPU movement and offered a number of options and libraries for a seamless experience for C, C++ and Fortran programmers



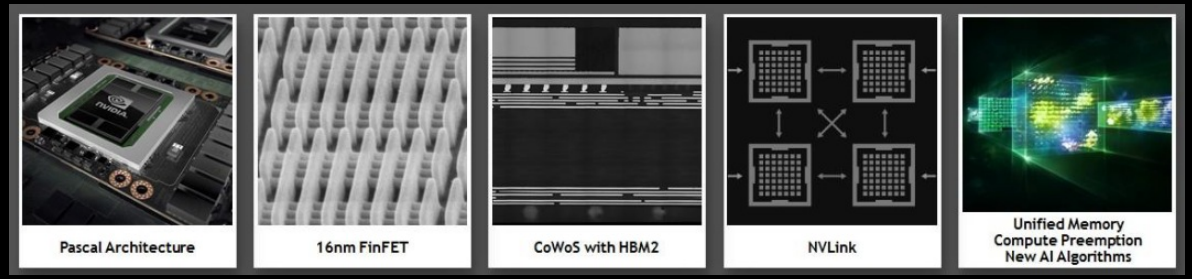


GPU Architecture

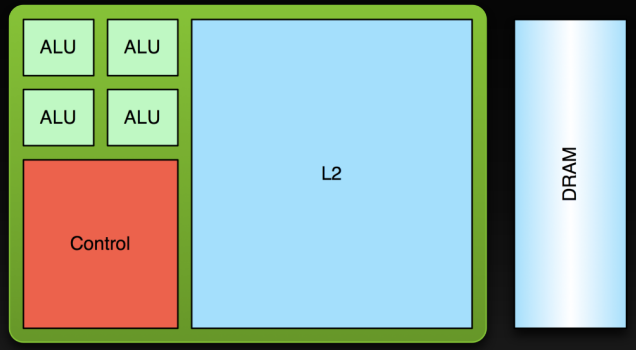


Pascal GPU

- A 15.3 billion transistor GPU in16nm FinFET
- NVLink™: High speed, high bandwidth interconnect for maximum application scalability
- HBM2: Fast, high capacity CoWoS (Chip-on-Wafer-on-Substrate) stacked memory architecture
- Unified Memory, Compute Preemption, and New AI Algorithms: Significantly improved programming model and advanced AI software optimization

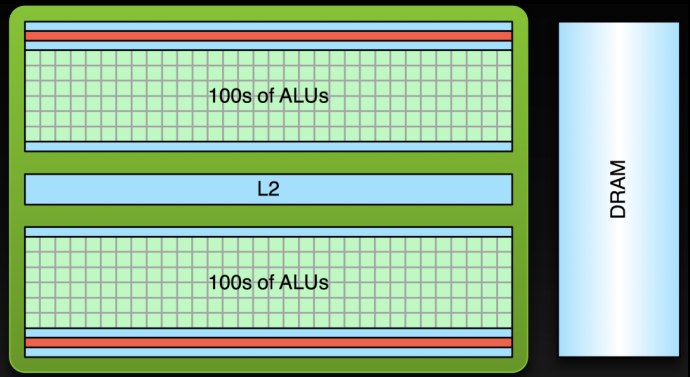


Different Philosophies of CPU and GPU



CPU

- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution

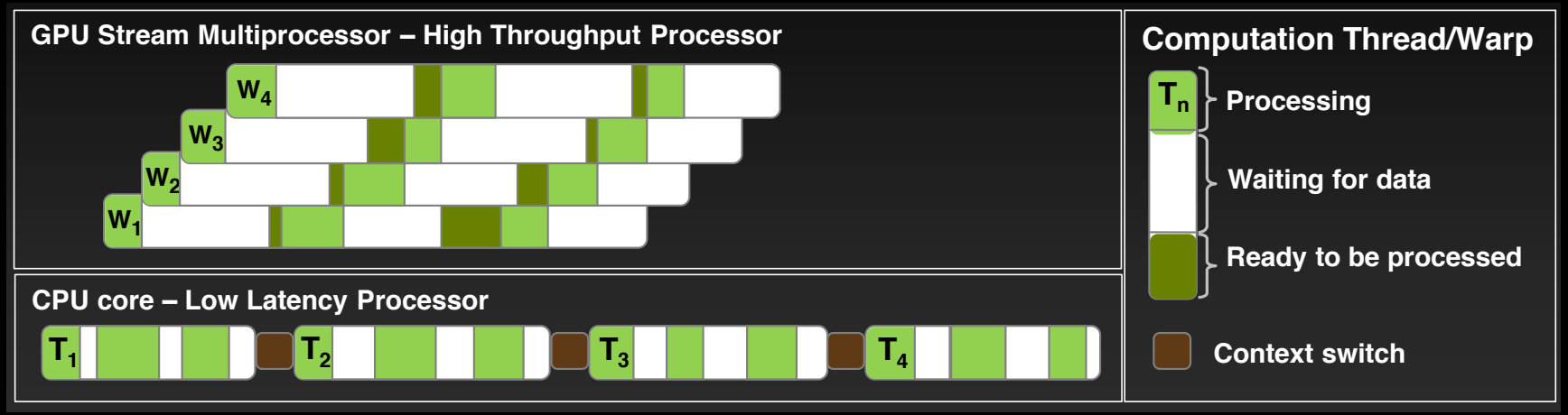


GPU

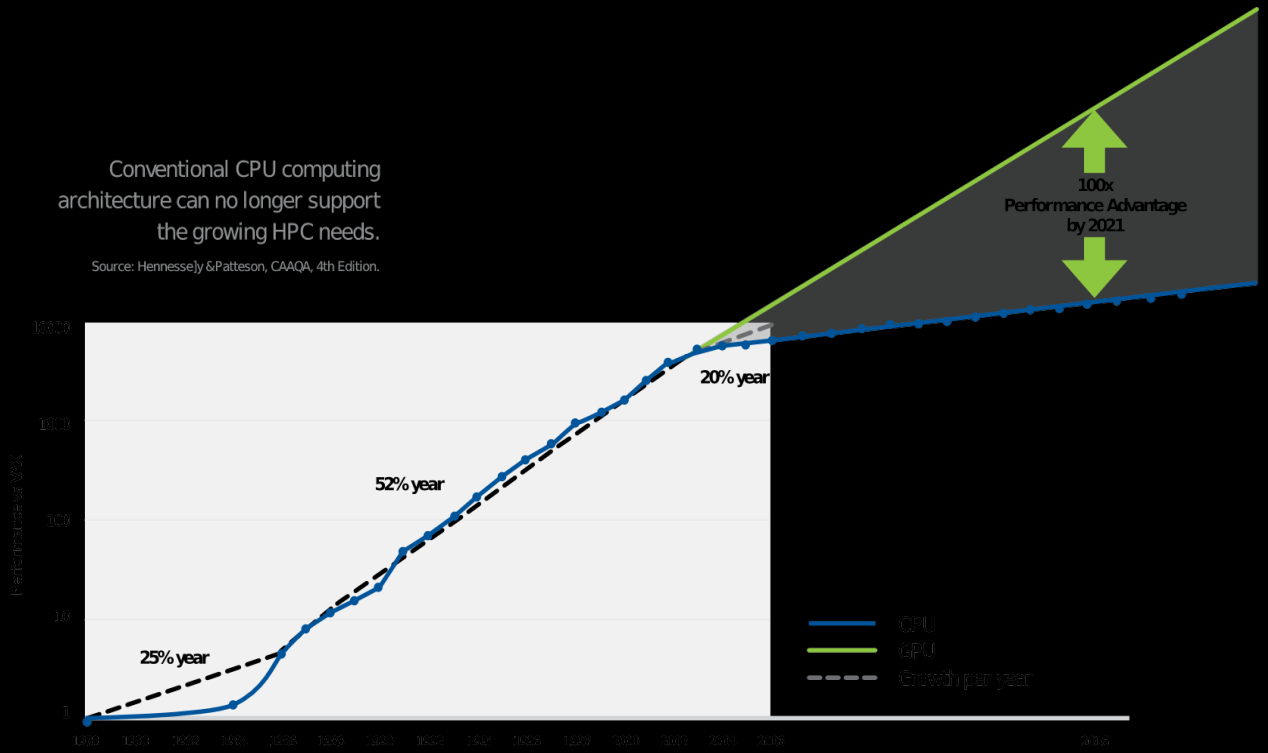
- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation

Different Philosophies of CPU and GPU

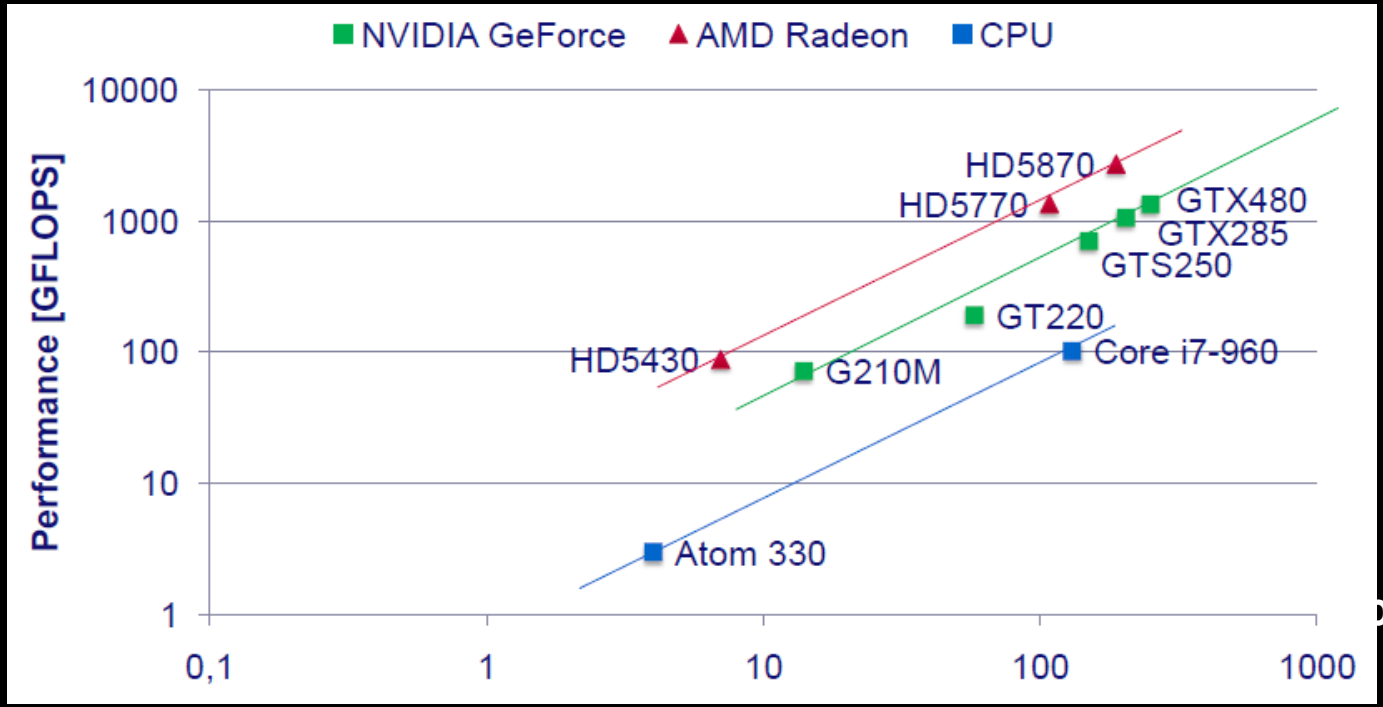
- CPU architecture must minimize latency within each thread
  GPU architecture hides latency with computation from other (warps of) threads



Performance Gap between CPU and GPU



Power Efficiency of CPU and GPU



提纲

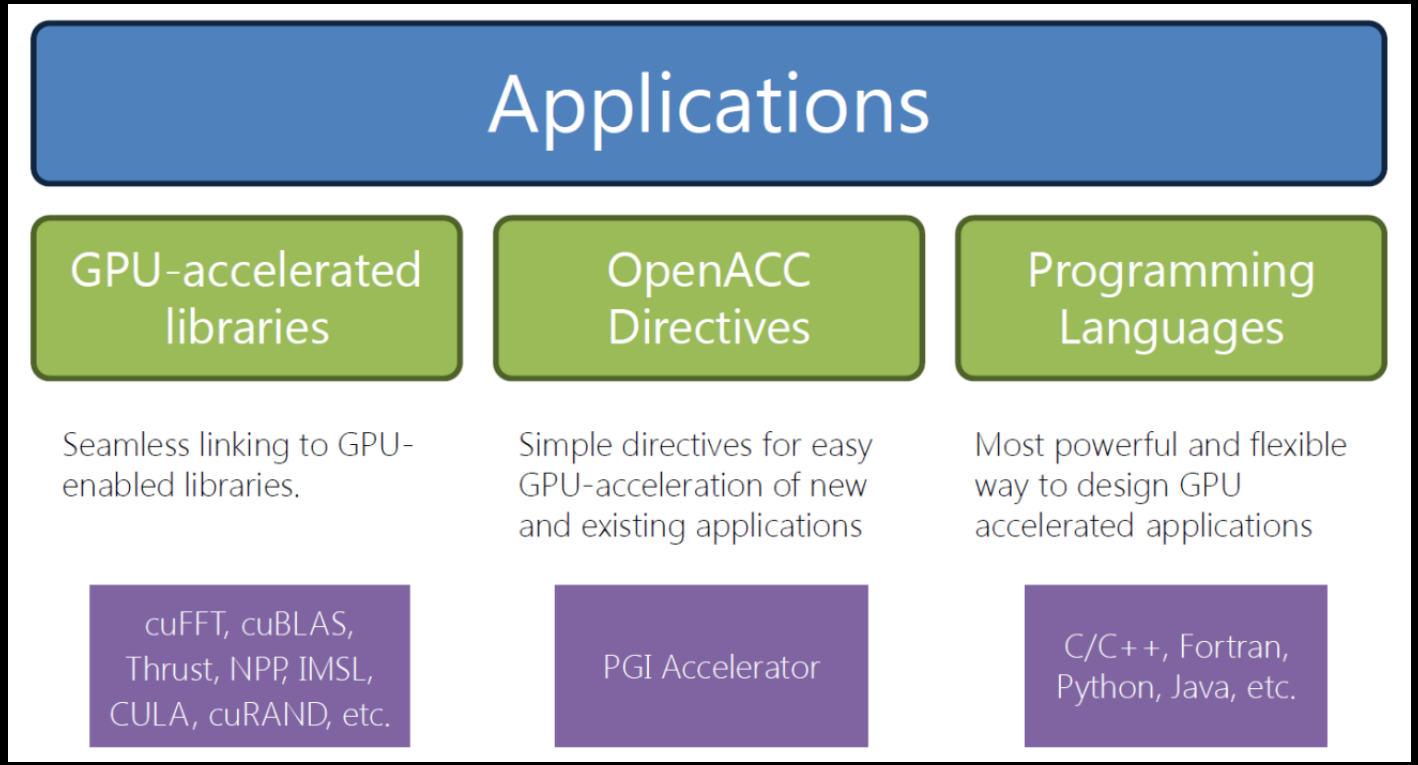
1. Road to GPU

2. CUDA Programming

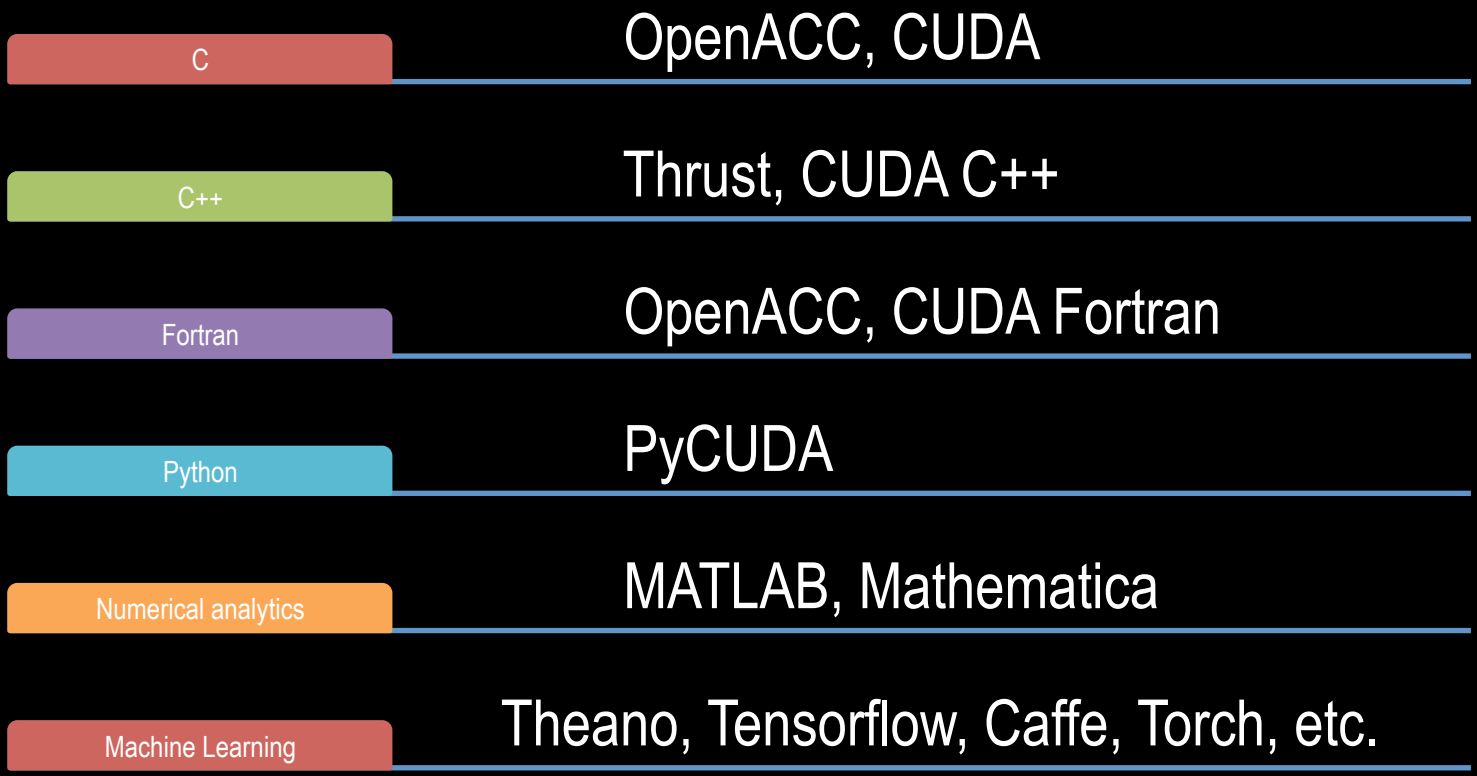
3. Code Optiization

4. GPU Machine Learning

GPU Programming Ecosystem

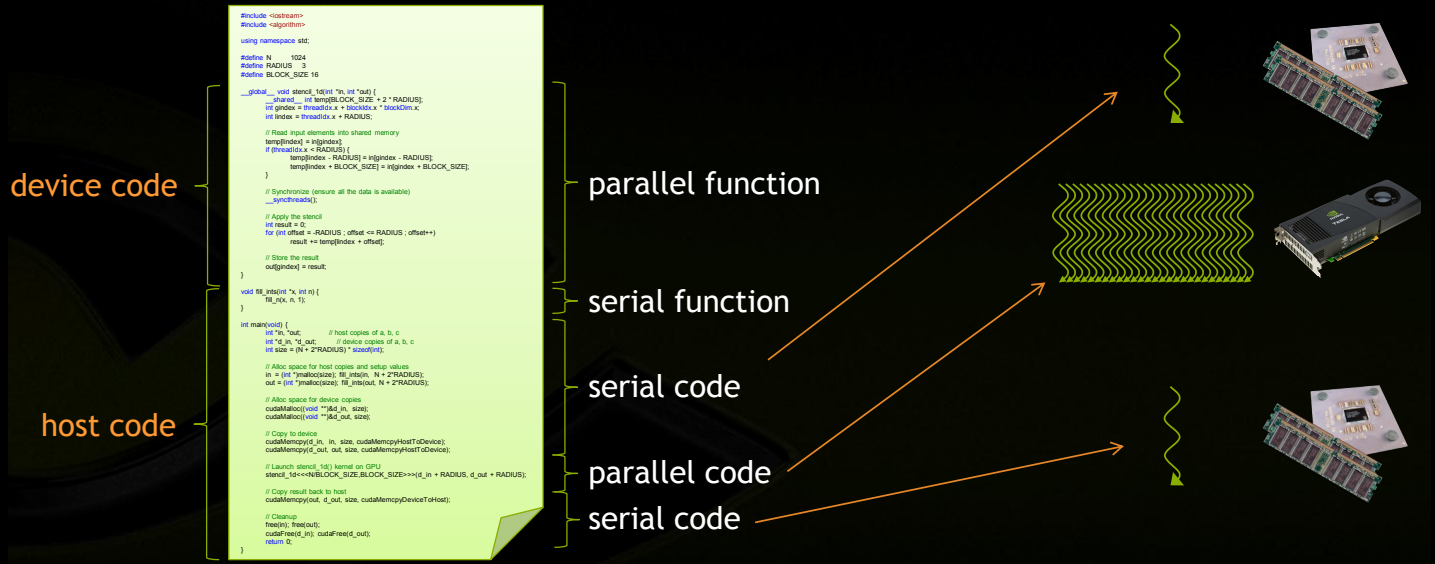


GPU Programming Tools

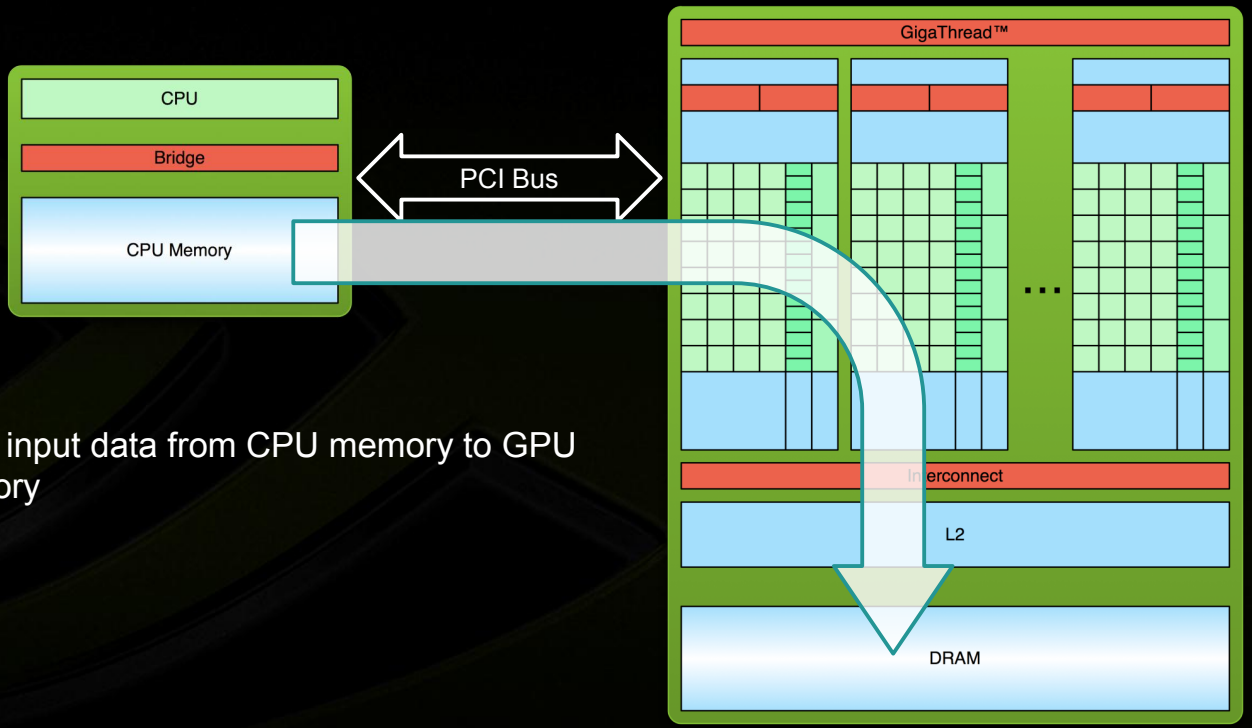


CUDA: Compute Unified Device Architecture

- NVIDIA's General Purpose Programing Model
- A massive number (>10000) of light-weight threads

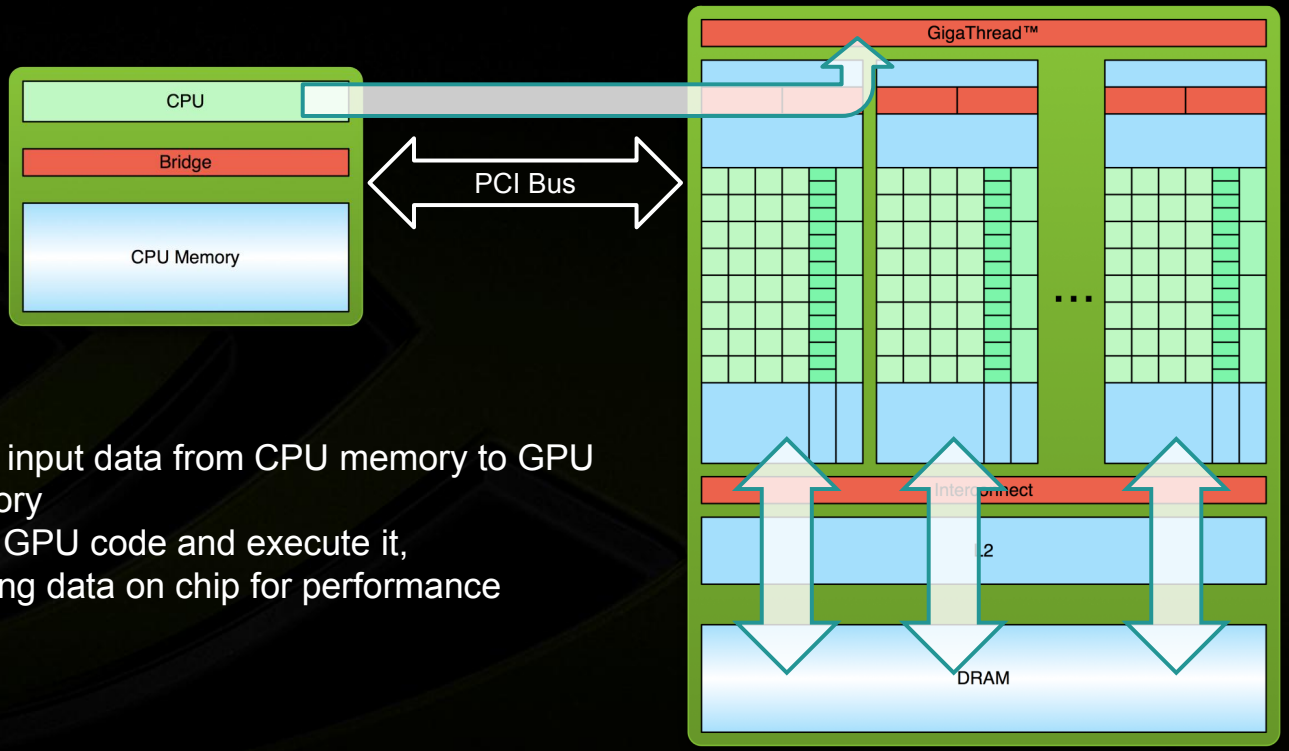


Processing Flow 1



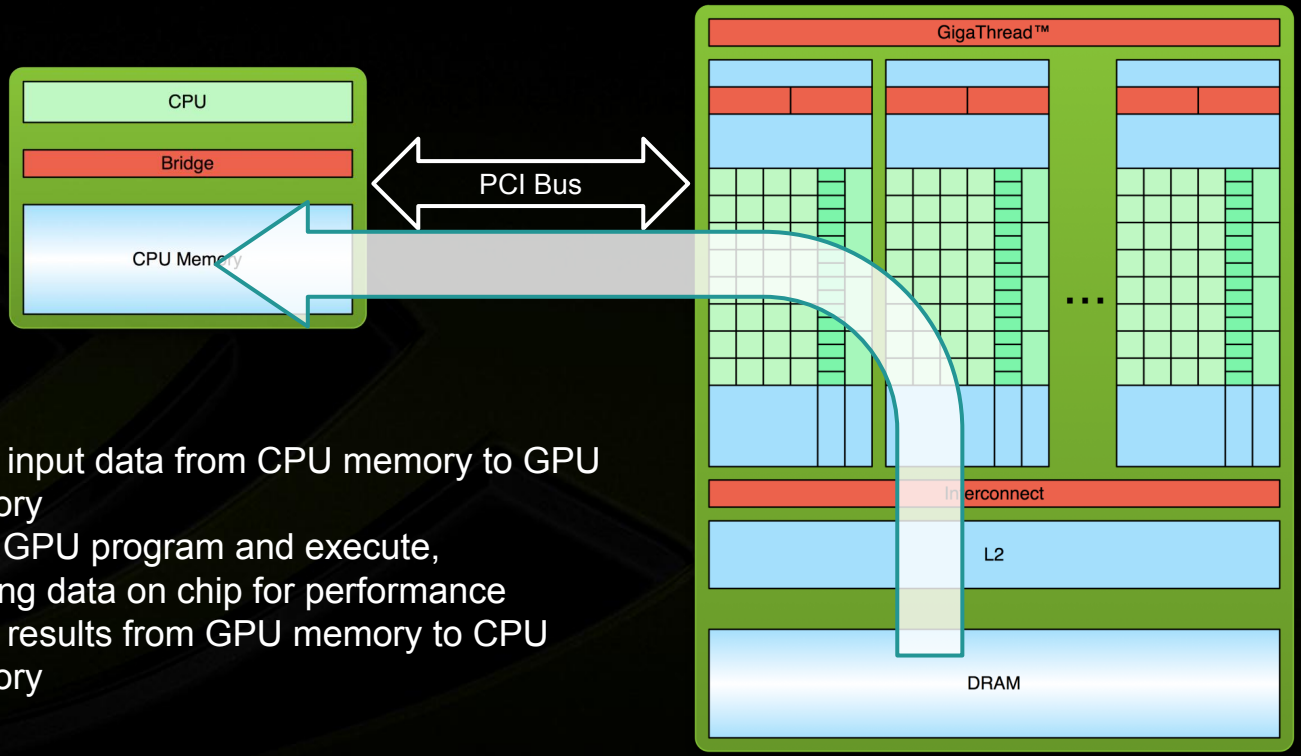
1. Copy input data from CPU memory to GPU memory

Processing Flow 2



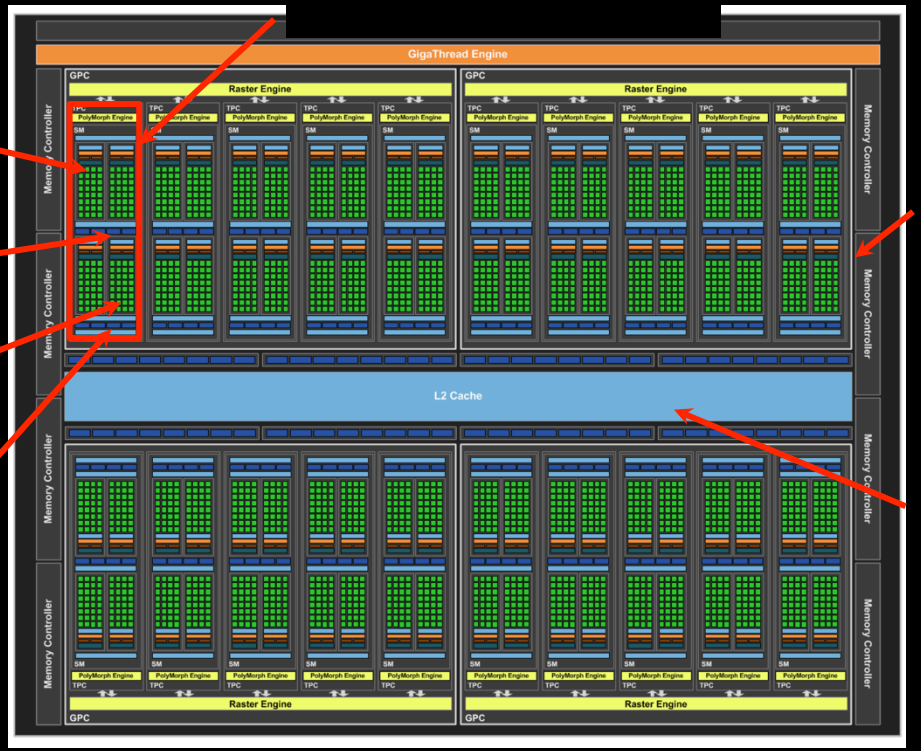
1. Copy input data from CPU memory to GPU memory
2. Load GPU code and execute it, caching data on chip for performance

Processing Flow 3



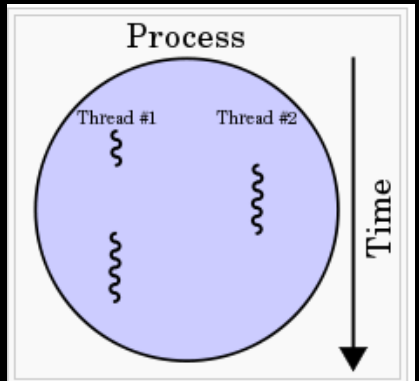
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Pascal GPU Architecture



Definition of Thread

- In computing, a process is an instance of a computer program that is being executed.
- Thread: a sequence of independently executed code
- In computer science, a thread of execution is the smallest unit of processing that can be scheduled by an operating system

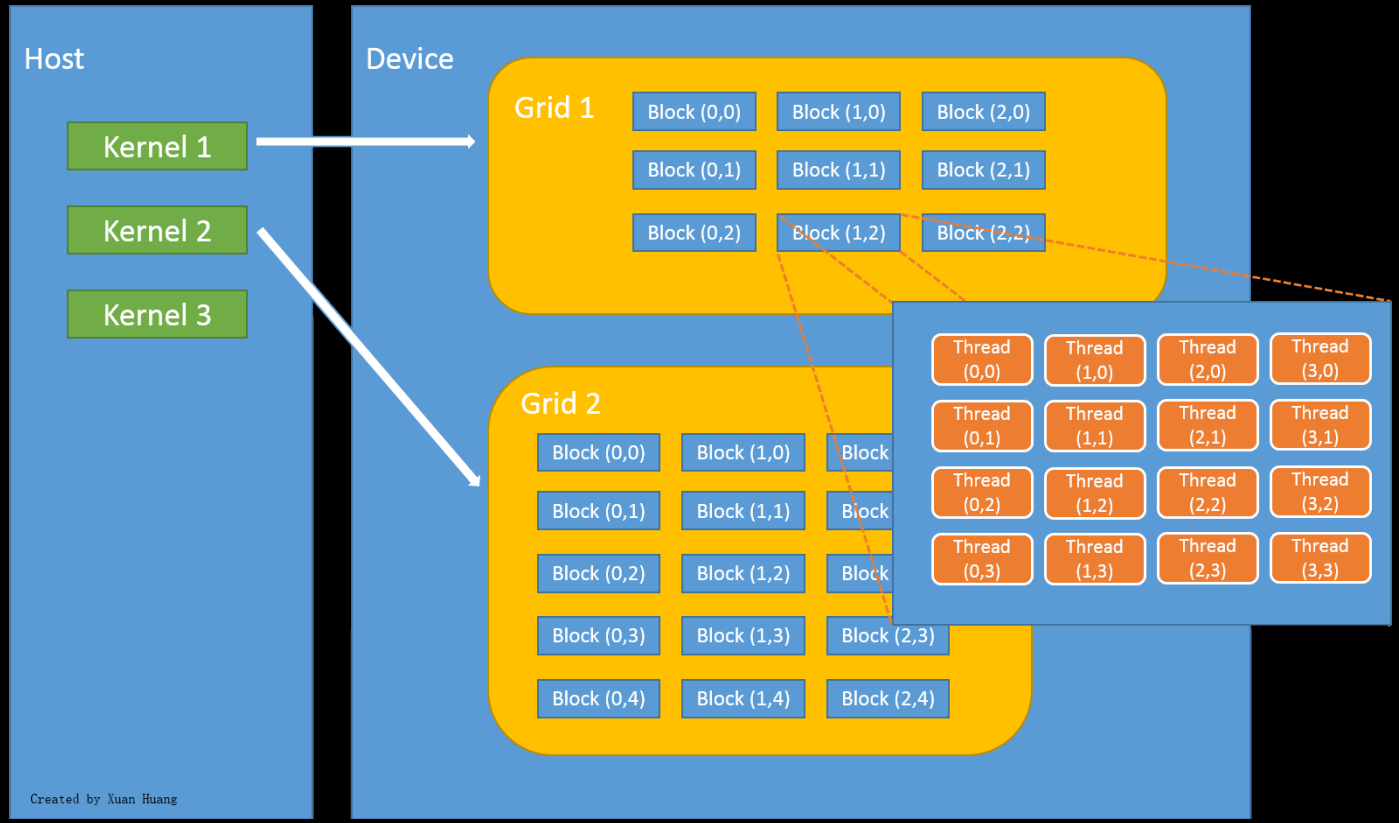


A process with two threads of execution on a single processor.

Definitions

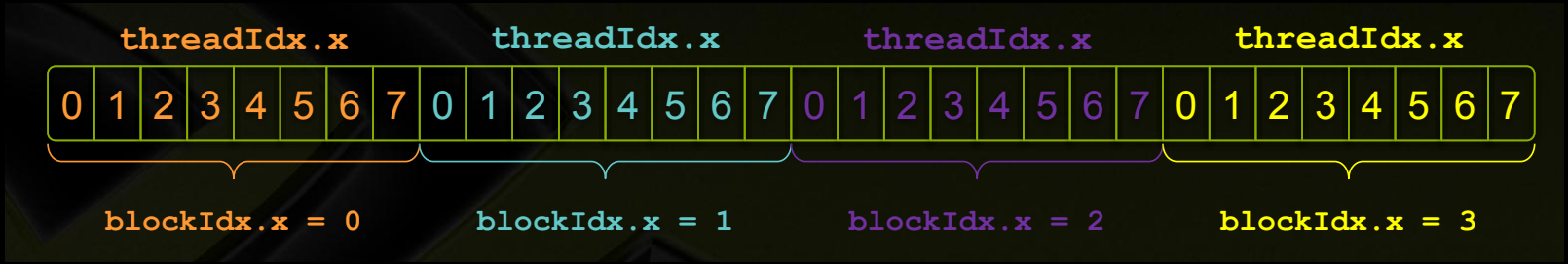
- *Thread*: concurrent code and associated state executed on the CUDA device (in parallel with other threads)
  - The unit of parallelism in CUDA
  - Note difference from CPU threads: creation cost, resource usage, and switching cost of GPU threads is much smaller
- *Warp*: a group of threads executed *physically* in parallel (SIMD)
- *Thread Block*: a group of threads that are executed together and can share memory on a single multiprocessor
- *Grid*: a group of thread blocks that execute a single CUDA program *logically* in parallel

Multithreading



Threading Indexing

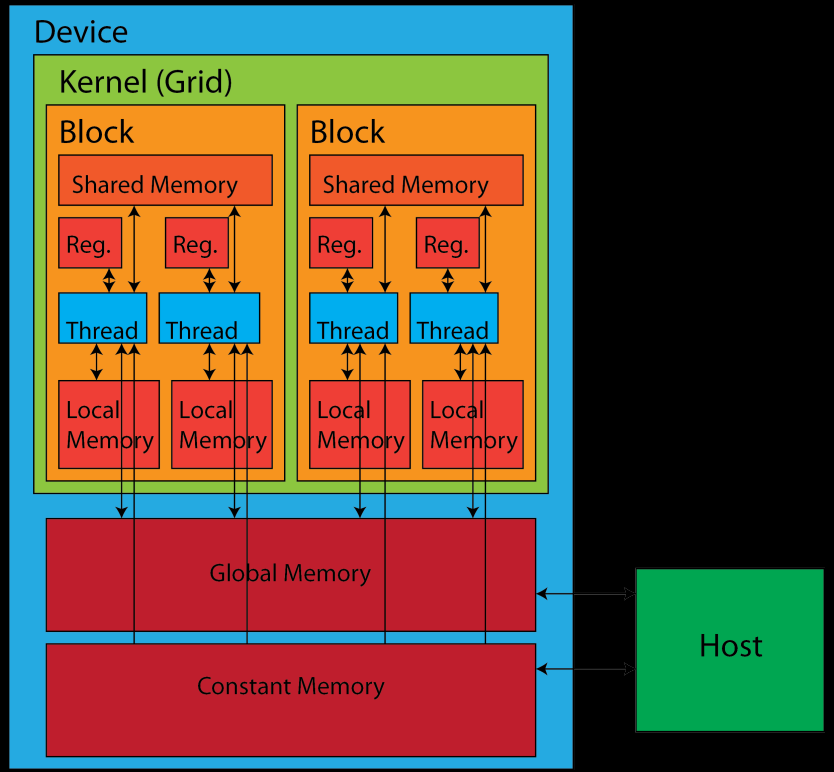
- Using blockIdx.x and threadIdx.x
  - Consider indexing an array with one element per thread (8 threads/block)



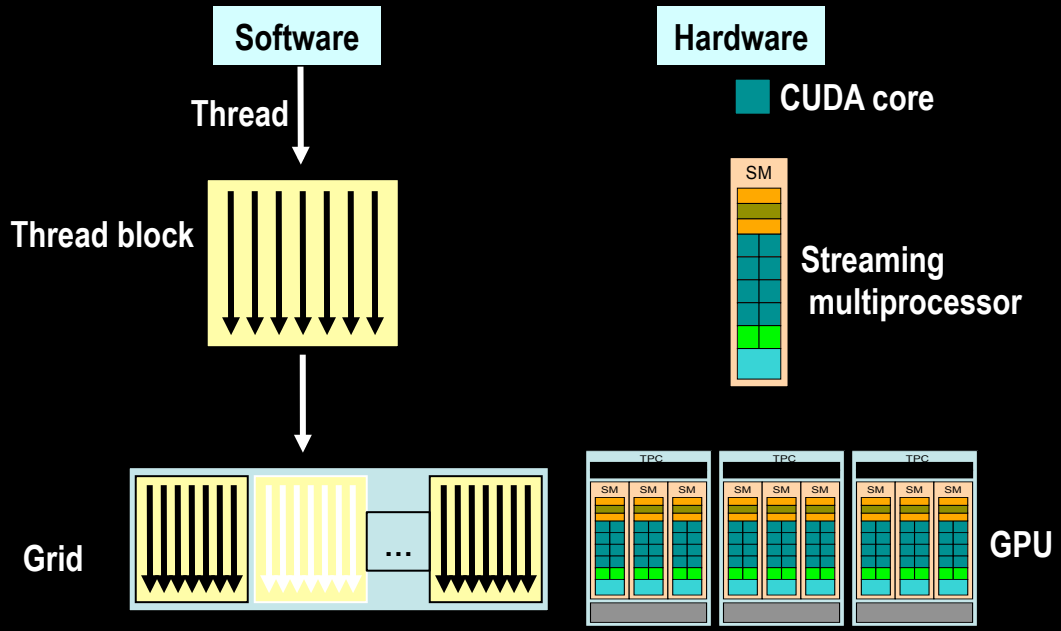
- With M threads per block, a unique index for each thread is given by:
  - int index = threadIdx.x + blockIdx.x * M;

CUDA Programming Model

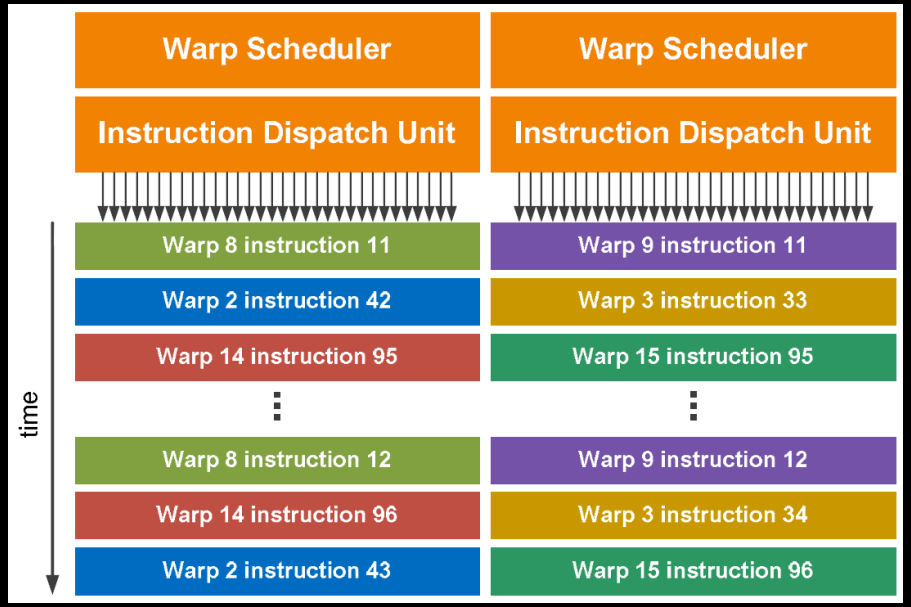
- Memory resources
  - GPU side
    - Register
    - Shared memory
    - Local memory
    - Global memory
    - Texture memory
    - Constant memory
  - CPU side
    - Main memory



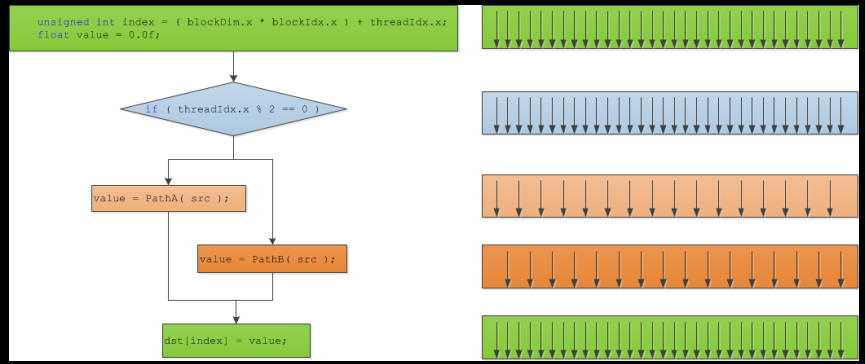
Parallel Program Organization in CUDA



CUDA Program Execution

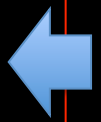



```
__global__ void TestDivergence( float* dst, float* src )
{
    unsigned int index = ( blockDim.x * blockIdx.x ) + threadIdx.x;
    float value = 0.0f;

    if ( threadIdx.x % 2 == 0 )
            value = PathA( src );
    else
            value = PathB( src );
    // Threads converge here again and execute in parallel.
    dst[index] = value;
}
```



CUDA: Hello, World! Example

```
/* Main function, executed on host (CPU) */
int main( void) {
        /* print message from CPU */
        printf( "Hello Cuda!\n" );

        /* execute function on device (GPU) */
        hello<<<NUM_BLOCKS, BLOCK_WIDTH>>>();

        /* wait until all threads finish their job */
        cudaDeviceSynchronize();

        /* print message from CPU */
        printf( "Welcome back to CPU!\n" );

        return(0);
}
```



Kernel:
A parallel function that runs on the GPU

```
/* Function executed on device (GPU */
__global__ void hello( void) {
        printf( "\tHello from GPU: thread %d and block %d\n",
                        threadIdx.x, blockIdx.x );
}
```

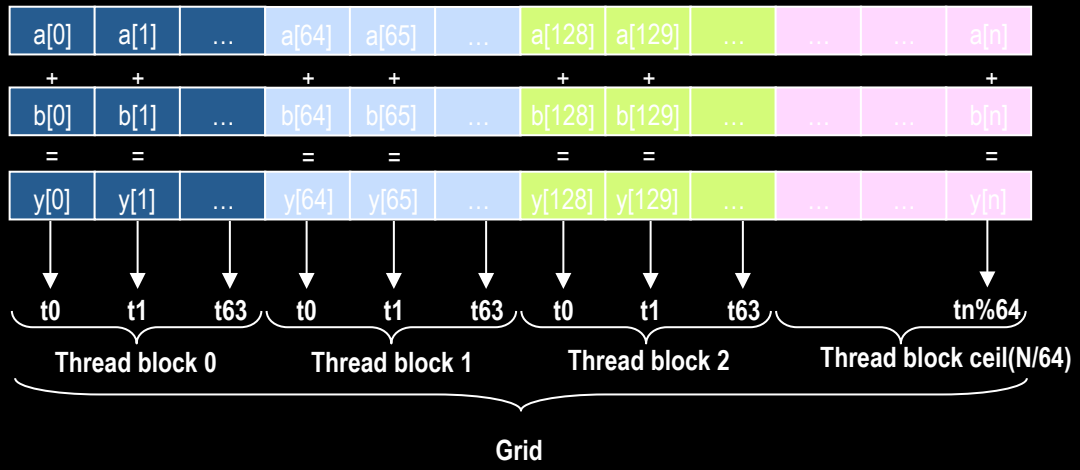
CUDA: Hello, World! Example

- Compile and build the program using NVIDIA's **nvcc** compiler:
  - **nvcc** -o helloCuda helloCuda.cu **-arch sm_20** Running the program on the GPU-enabled node:
- **helloCuda**
  - Hello Cuda!
    - Hello from GPU: thread 0 and block 0
    - Hello from GPU: thread 1 and block 0
    - . . .
    - Hello from GPU: thread 6 and block 2
    - Hello from GPU: thread 7 and block 2
  - Welcome back to CPU!

```
/* Main function, executed on host (CPU) */
int main( void) {
        /* print message from CPU */
        printf( "Hello Cuda!\n" );

        /* execute function on device (GPU) */
        hello<<<NUM_BLOCKS, BLOCK_WIDTH>>>();

        /* wait until all threads finish their job */
        cudaDeviceSynchronize();

        /* print message from CPU */
        printf( "Welcome back to CPU!\n" );

        return(0);
}
```

CUDA: Vector Addition Example

- Vector addition (N elements/vector)
  - 1 thread for one addition
  - 64 threads per block
  - ceil(N/64) thread blocks



CUDA: Vector Addition Example

```
/* Main function, executed on host (CPU) */
int main( void) {
    /* 1. allocate memory on GPU */
    /* 2. Copy data from Host to GPU */
    /* 3. Execute GPU kernel */
    /* 4. Copy data from GPU back to Host */
    /* 5. Free GPU memory */
    return(0);
}
```

CUDA: Vector Addition Example

```
/* 1. allocate memory on GPU */
float *d_A = NULL;
if (cudaMalloc((void **)&d_A, size) != cudaSuccess)
        exit(EXIT_FAILURE);

float *d_B = NULL;
cudaMalloc((void **)&d_B, size); /* For clarity we'll not check for err */

float *d_C = NULL;
cudaMalloc((void **)&d_C, size); intro
```

CUDA: Vector Addition Example

```
/* 2. Copy data from Host to GPU */
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

CUDA: Vector Addition Example

```
/* 3. Execute GPU kernel */
/* Calculate number of blocks and threads */
int threadsPerBlock = 64;
int blocksPerGrid =(numElements + threadsPerBlock - 1) / threadsPerBlock;

/* Launch the Vector Add CUDA Kernel */
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);

/* Wait for all the threads to complete */
cudaDeviceSynchronize();
```

CUDA: Vector Addition Example

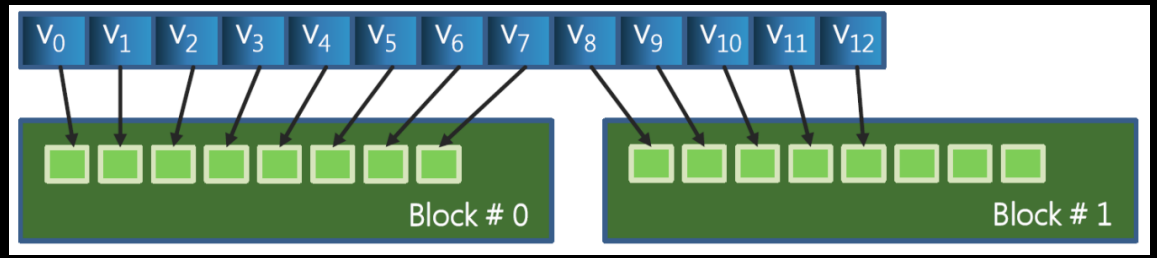
```
/* 4. Copy data from GPU back to Host */
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```

CUDA: Vector Addition Example

```
/* 5. Free GPU memory */
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

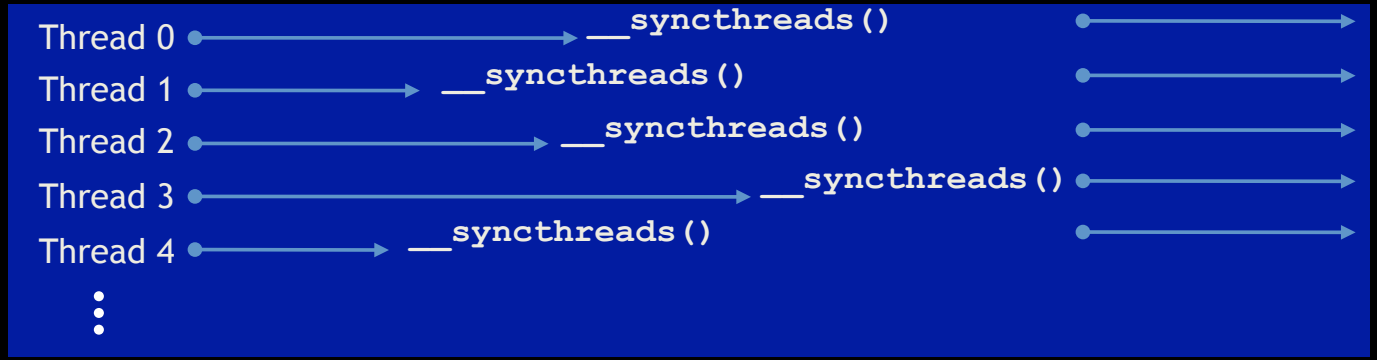
CUDA: Vector Addition Example

```
/* CUDA Kernel */
 __global__ void vectorAdd( const float *A, const float *B, float *C,
                                                int numElements) {
      /* Calculate the position in the array */
      int i = blockDim.x * blockIdx.x + threadIdx.x;
      /* Add 2 elements of the array */
      if (i < numElements) C[i] = A[i] + B[i];
}
```



Concurrency Management

- We can synchronize threads with function __syncthreads()
- Threads in the block wait until *all* threads have hit the __syncthreads()
  - Threads are *only* synchronized within a block!

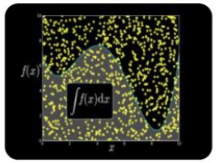


Parallel Dot Product: dot()

```
__global__ void dot( int *a, int *b, int *c ) {
        __shared__ int temp[N];
        temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

        __syncthreads();

        if( 0 == threadIdx.x ) {
                int sum = 0;
                for( int i = N-1; i >= 0; i-- )
                        sum += temp[i];

        *c = sum; }
}
```

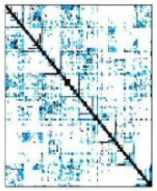
GPU Accelerated Libraries

NVIDIA
cuBLAS



NVIDIA cuRAND

NVIDIA NPP



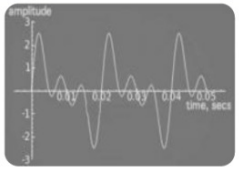
NVIDIA cuSPARSE

CUSP

Sparse
Linear
Algebra

Thrust

C++ STL
Features for
CUDA



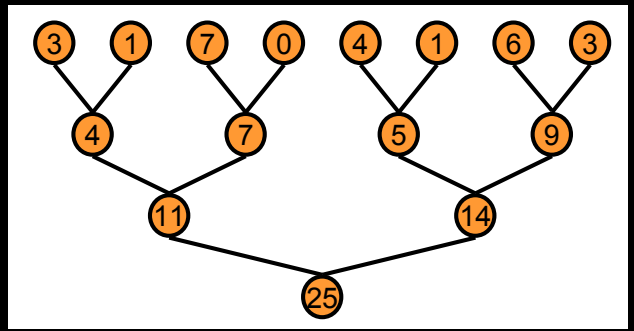
NVIDIA cuFFT

提纲

1. Road to GPU
2. CUDA Programming
3. Code Optimization
4. GPU Machine Learning

Problem: Reduction

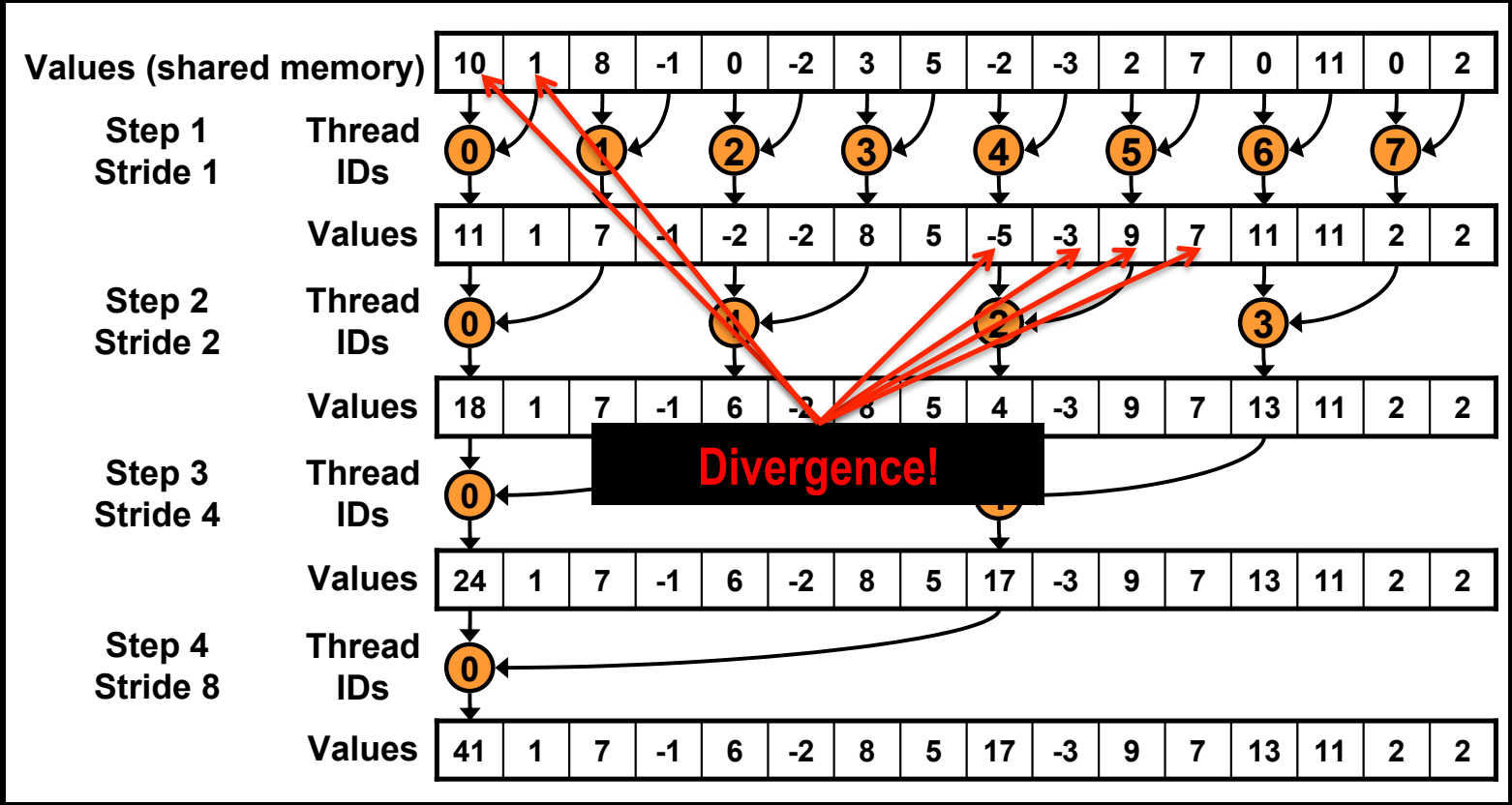
- Given an array a[], compute a[0]⊙a[1]⊙a[2]⊙…⊙a[n-1], where ⊙ is an arbitrary binary math operator
  - e.g., sum(a) = a[0] + a[1] + … + a[n-1]
- Tree-based approach used within each thread block



Reduction #1: Interleaved Addressing

```
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Reduction #1: Interleaved Addressing



Reduction #2: Interleaved Addressing

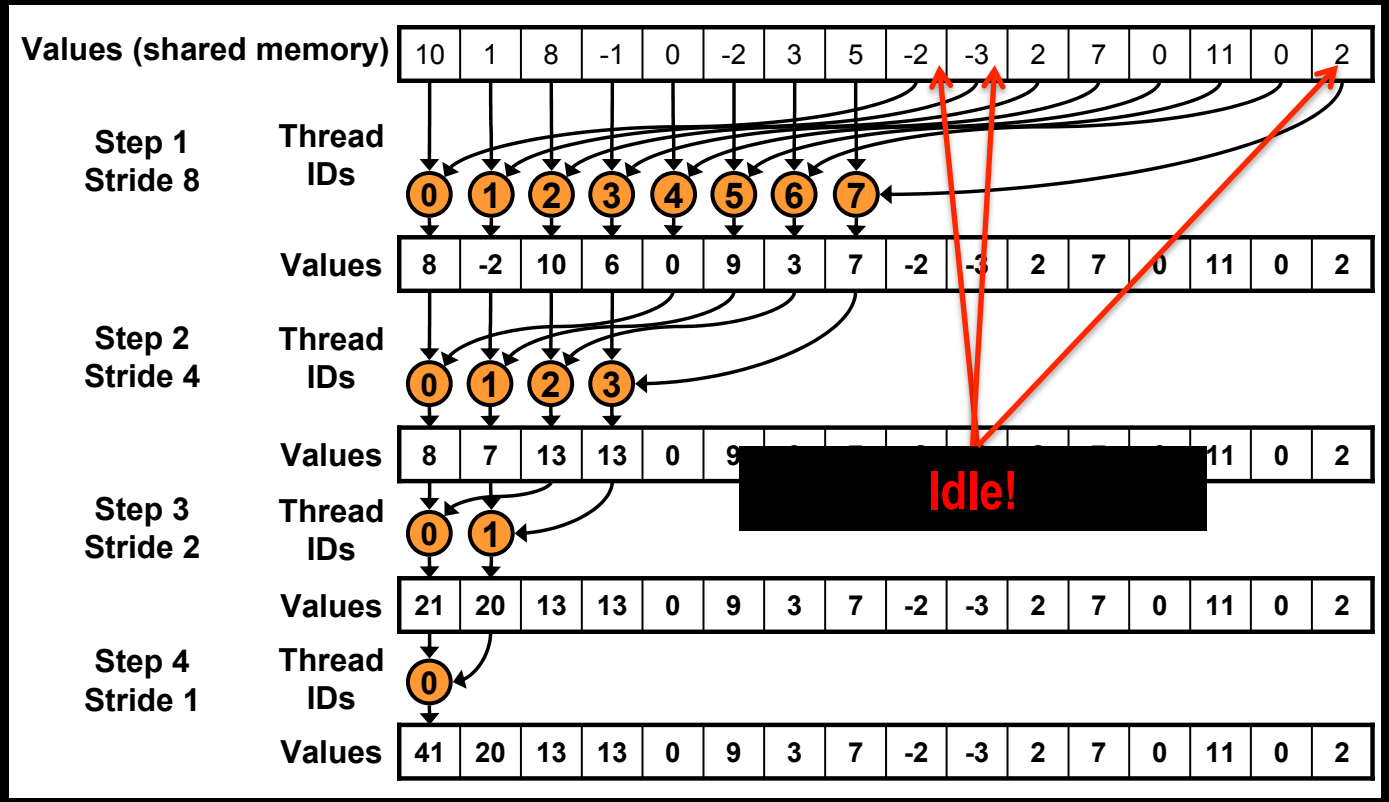
Just replace divergent branch in inner loop:

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

With strided index and non-divergent branch:

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

Reduction #2: Interleaved Addressing



Reduction #3 & 4: First Add During Load

Halve the number of blocks, and replace single load:

```
// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

With two loads and first add of the reduction:

```
// perform first level of reduction,
// reading from global memory, writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

Reduction #5: Unroll the Last Warp

```
__global__ void reduce0(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];

  // each thread loads one element from global to shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[tid] = g_idata[i];
  __syncthreads();

  // do reduction in shared mem
  for(unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }

  // write result for this block to global mem
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

```
for (unsigned int s=blockDim.x/2; s>32; s>>=1)
{
  if (tid < s)
    sdata[tid] += sdata[tid + s];
  __syncthreads();
}

if (tid < 32)
{
  sdata[tid] += sdata[tid + 32];
  sdata[tid] += sdata[tid + 16];
  sdata[tid] += sdata[tid +  8];
  sdata[tid] += sdata[tid +  4];
  sdata[tid] += sdata[tid +  2];
  sdata[tid] += sdata[tid +  1];
}
```

Reduction #6: Reduction #4: Unroll the Last Warp

```
if (blockSize >= 512) {
    if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads();
}
if (blockSize >= 256) {
    if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads();
}
if (blockSize >= 128) {
    if (tid <  64)  { sdata[tid] += sdata[tid +   64]; } __syncthreads();
}

if (tid < 32) {
    if (blockSize >=  64) sdata[tid] += sdata[tid + 32];
    if (blockSize >=  32) sdata[tid] += sdata[tid + 16];
    if (blockSize >=  16) sdata[tid] += sdata[tid +  8];
    if (blockSize >=   8)  sdata[tid] += sdata[tid +  4];
    if (blockSize >=   4)  sdata[tid] += sdata[tid +  2];
    if (blockSize >=   2)  sdata[tid] += sdata[tid +  1];
}
```

Performance

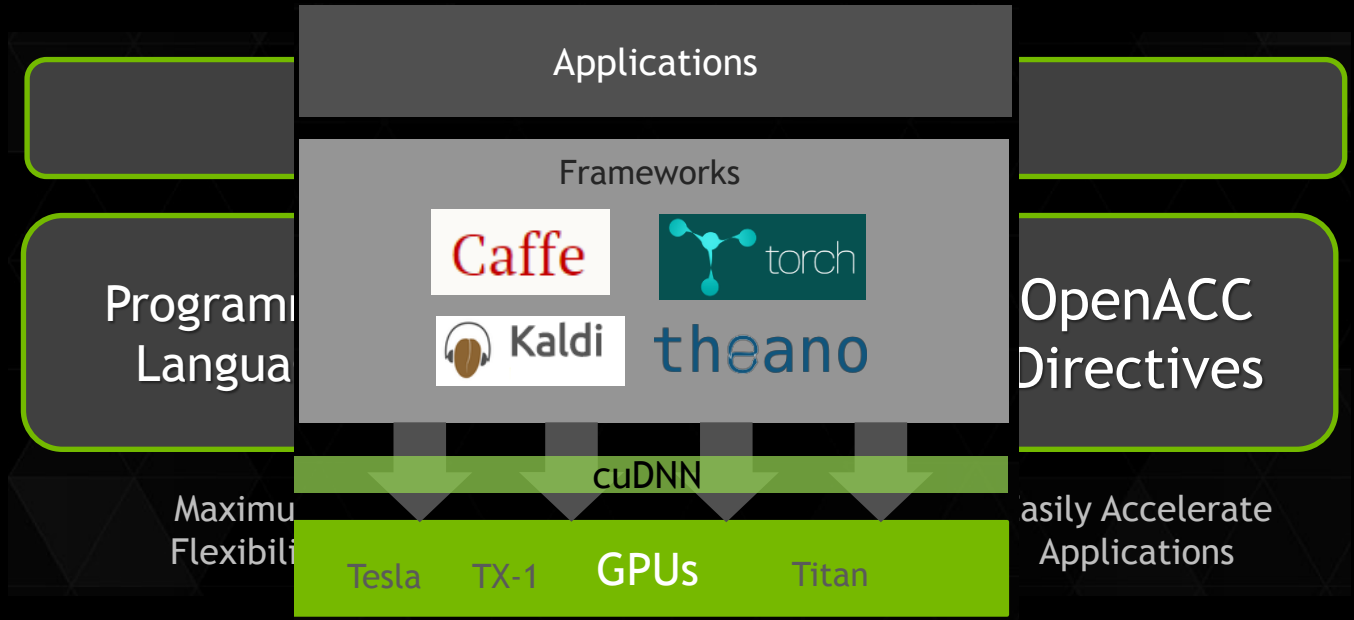
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x

提纲

1. Road to GPU
2. CUDA Programming
3. Code Optimization
4. GPU Machine Learning

cuDNN

- cuDNN is a library of primitives for deep learning



cuDNN

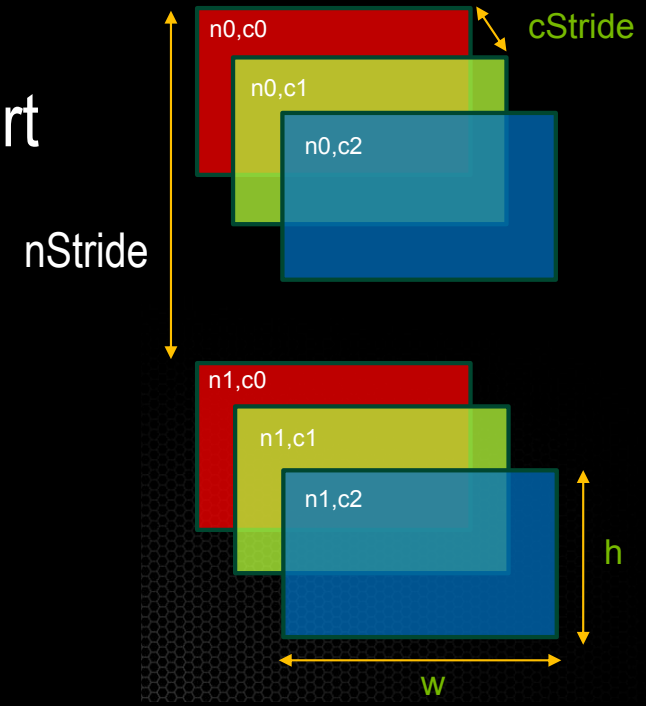
- Low-level Library of GPU-accelerated routines; similar in intent to BLAS
- Out-of-the-box speedup of Neural Networks
- Developed and maintained by NVIDIA
- Optimized for current and future NVIDIA GPU generations
- First release focused on Convolutional Neural Networks

cuDNN Features

- Flexible API : arbitrary dimension ordering, striding, and sub-regions for 4d tensors
- Less memory, more performance : Efficient forward and backward convolution routines with zero memory overhead
- Easy Integration : black box implementation of convolution and other routines – ReLu, Sigmoid, Tanh, Pooling, Softmax

Tensor-4d

- Image Batches described as 4D Tensor [n, c, h, w] with stride support [nStride, cStride, hStride, wStride]
- Allows flexible data layout
- Easy access to subsets of features (Caffe's "groups")
- Implicit cropping of sub-images



Common Data Structure for Layer

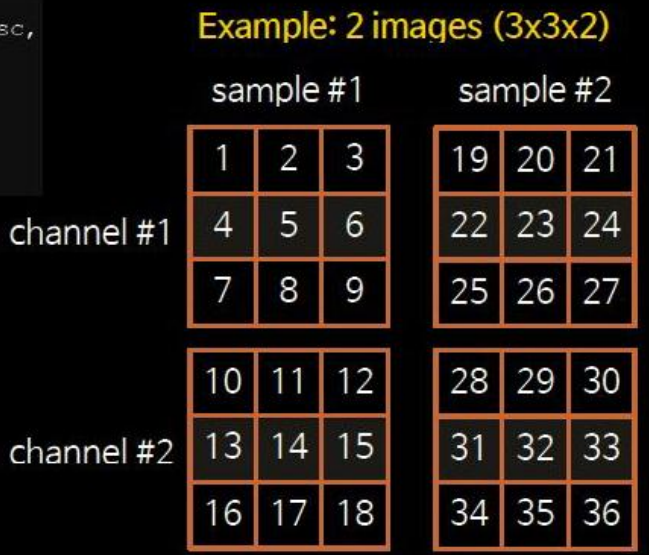
- Device memory & tensor description for input/output data & error Tensor Description defines dimensions of data
- Float(*d_input, *d_output, *d_inputDelta, *d_outputDelta
  cudnnTensorDescriptor_t inputDesc;
  cudnnTensorDescriptor_t outputDesc)

Data Layer

- Create and set tensor descriptor

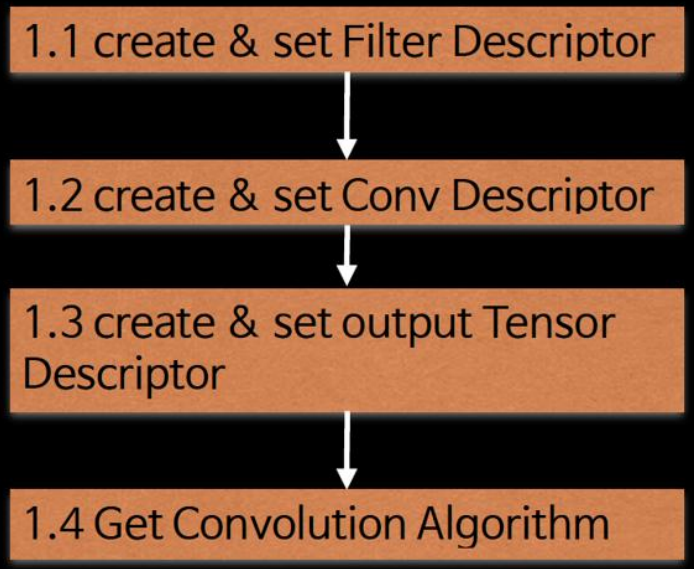
```
cudnnStatus_t
cudnnSetTensor4dDescriptor( cudnnTensorDescriptor_t    tensorDesc,
                            cudnnTensorFormat_t format,
                            cudnnDataType_t dataType,
                            int n,
                            int c,
                            int h,
                            int w )
```

```
cudnnSetTensor4dDescriptor(
                        outputDesc,
                        CUDNN_TENSOR_NCHW,
                        CUDNN_FLOAT,
                        sampleCnt,
                        channels,
                        height,
                        width
);
```



Convolution Layer

- 1.Initialization




```
cudnnCreateFilterDescriptor(&filterDesc);
cudnnSetFilter4dDescriptor(…);

cudnnCreateConvolutionDescriptor(&convDesc);
cudnnSetConvolution2dDescriptor(…);

cudnnGetConvolution2dForwardOutputDim(…);
cudnnCreateTensorDescriptor(&dstTensorDesc);
cudnnSetTensor4dDescriptor();

cudnnGetConvolutionForwardAlgorithm(…);
cudnnGetConvolutionForwardWorkspaceSize(…);
```

Convolution Layer

- 2.Convolution

```
cudnnStatus_t
cudnnConvolutionForward( cudnnHandle_t                    handle,
                         const void                      *alpha,
                         const cudnnTensorDescriptor_t    srcDesc,
                         const void                      *srcData,
                         const cudnnFilterDescriptor_t    filterDesc,
                         const void                      *filterData,
                         const cudnnConvolutionDescriptor_t  convDesc,
                         cudnnConvolutionFwdAlgo_t        algo,
                         void                            *workSpace,
                         size_t
 workSpaceSizeInBytes,
                         const void                      *beta,

                         const cudnnTensorDescriptor_t    destDesc,
                         void                            *destData )
```

d_input
inputDesc

d_output
outputDesc

Convolution Layer

- 3.Activation

```
cudnnStatus_t
cudnnActivationForward( cudnnHandle_t                    handle,
                        cudnnActivationMode_t            mode,
                        const void                      *alpha,
                        const cudnnTensorDescriptor_t    srcDesc,
                        const void                      *srcData,
                        const void                      *beta,
                        const cudnnTensorDescriptor_t    destDesc,
                        void                            *destData )
```

sigmoid
tanh
ReLU

outputDesc
d_output

Convolution Layer

- 4. Backward pass

Pooling and Softmax

Pooling Layer

1. Initilization	cudnnCreatePoolingDescriptor(&poolingDesc); cudnnSetPooling2dDescriptor(…);
2. Forward Pass	cudnnPoolingForward(…);
3. Backward Pass	cudnnPoolingBackward(…);

Softmax Layer

Forward Pass	cudnnSoftmaxForward(…);

Training

- Multi-GPU in general
  - Data parallelization vs. model parallelization
    - Data parallelization: Distribute the data, use the same model
    - Model parallelization: Distribute the model, use the same data
  - Data parallelization & gradient average
    - One of the easiest way to use multi-GPU
    - The same results as the case of single GPU

Example – OverFeat Layer 1

```
/* Allocate memory for Filter and ImageBatch, fill with data */
cudaMalloc( &ImageInBatch , ... );
cudaMalloc( &Filter , ... );
...
 /* Set descriptors */
cudnnSetTensor4dDescriptor( InputDesc, CUDNN_TENSOR_NCHW, 128, 96, 221, 221);
cudnnSetFilterDescriptor( FilterDesc, 256, 96, 7, 7 );
cudnnSetConvolutionDescriptor( convDesc, InputDesc, FilterDesc, pad_x, pad_y, 2, 2, 1, 1, CUDNN_CONVOLUTION);

/* query output layout */
cudnnGetOutputTensor4dDim(convDesc, CUDNN_CONVOLUTION_FWD, &n_out, &c_out, &h_out, &w_out);

/* Set and allocate output tensor descriptor */
cudnnSetTensor4dDescriptor( &OutputDesc, CUDNN_TENSOR_NCHW, n_out, c_out, h_out, w_out); cudaMalloc(&ImageBatchOut, n_out *
c_out * h_out * w_out * sizeof(float));

/* launch convolution on GPU */
cudnnConvolutionForward( handle, InputDesc, ImageInBatch, FilterDesc, Filter, convDesc, OutputDesc, ImageBatchOut,
CUDNN_RESULT_NO_ACCUMULATE);
```

Implementation 1: 2D conv as a GEMV

Image

I1	I2	I3	I4	I5	I6
I7	I8	I9	I10	I11	I12
I13	I14	I15	I16	I17	I18
I19	I20	I21	I22	I23	I24
I25	I26	I27	I28	I29	I30
I31	I32	I33	I34	I35	I36

F1	F2	F3
F4	F5	F6
F7	F8	F9

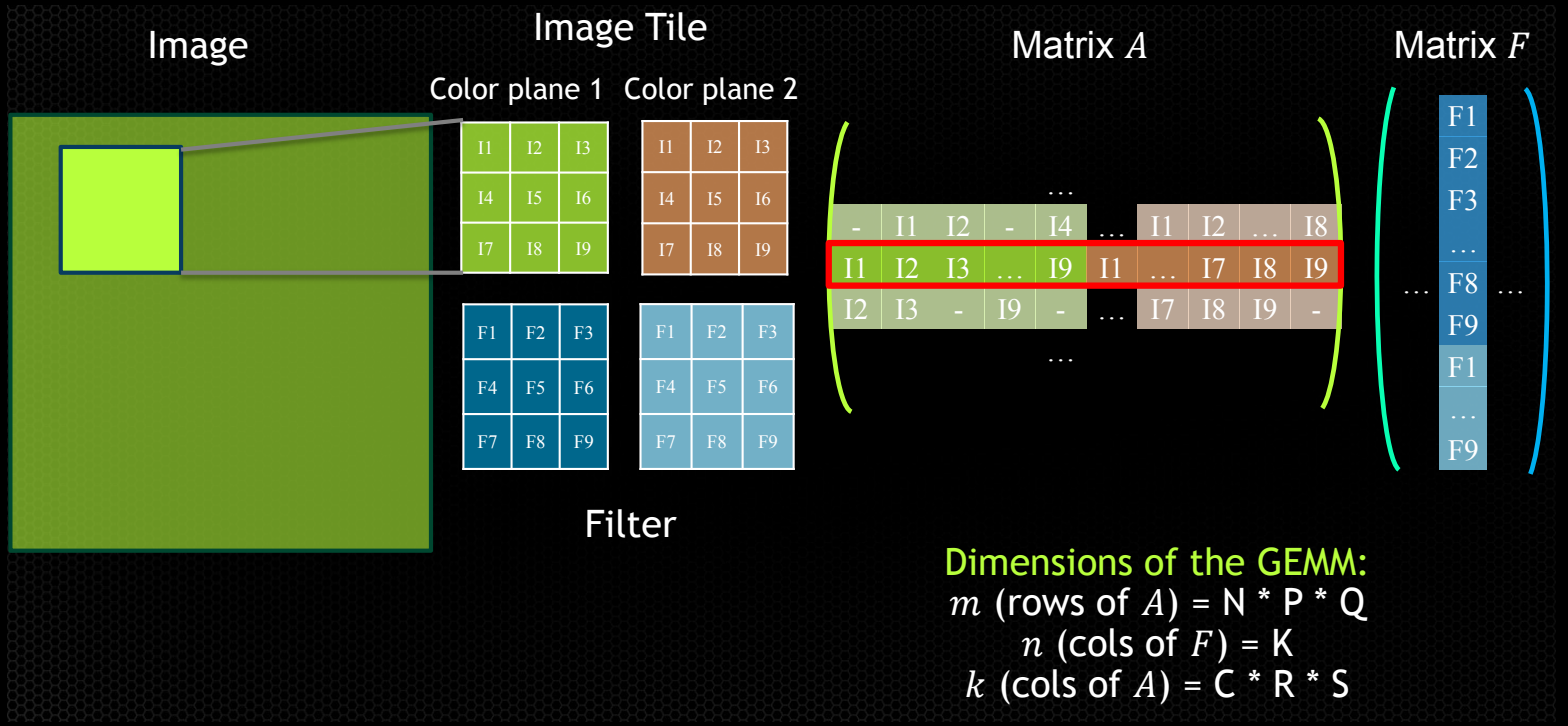
Filter

I1	I2	I3	I7	I8	I9	I13	I14	I15
I2	I3	I4	I8	I9	I10	I14	I15	I16
I3	I4	I5	I9	I10	I11	I15	I16	I17

F1
F2
F3
F4
F5
F6
F7
F8
F9

A lot of data duplication!

Implementation 2: Multi-convolve as GEMM



Dimensions of the GEMM:

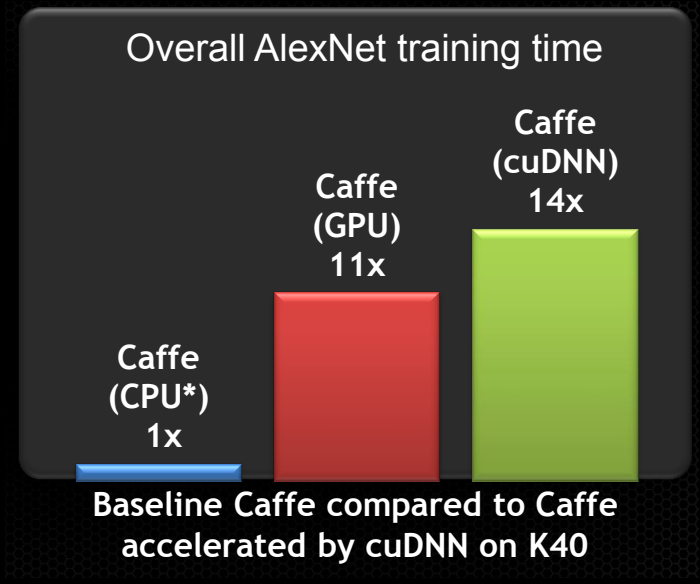
m (rows of A) = N * P * Q

n (cols of F) = K

k (cols of A) = C * R * S

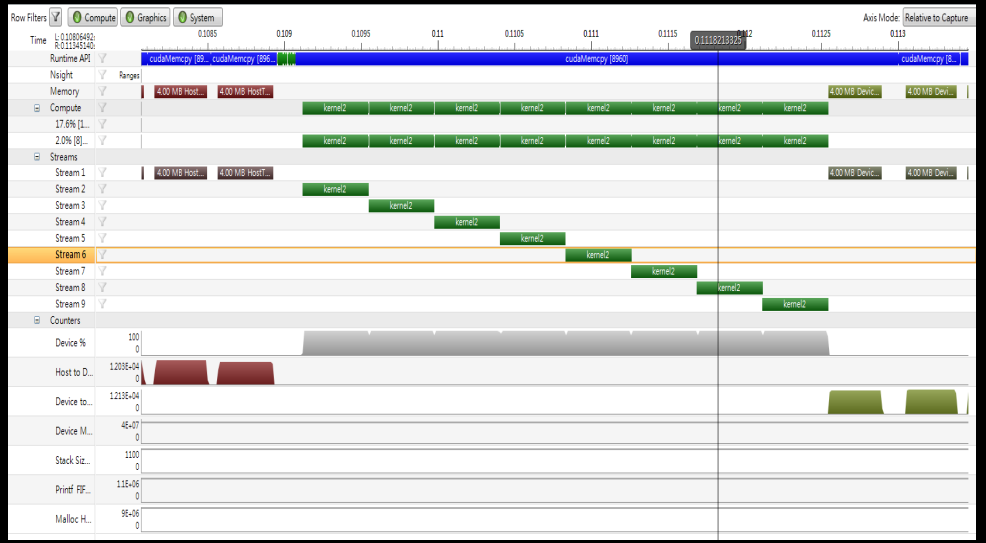
Using Caffe with cuDNN

- Accelerate Caffe layer types by 1.2 – 3x
- On average, 36% faster overall for training on Alexnet
- Integrated into Caffe dev branch today!
  (official release with Caffe 1.0)
- Seamless integration with a global switch



Development Environment

- NSight IDE
  - Linux, Mac & Windows
  - GPU Debugging and profiling
- CUDA-GDB debugger
- NVIDIA Visual Profiler



CUDA Resources

- CUDA and CUDA libraries examples: http://docs.nvidia.com/cuda/cuda-samples/;
- NVIDIA's Cuda Resources: https://developer.nvidia.com/cuda-education
- CUDA C/C++ & Fortran: http://developer.nvidia.com/cuda-toolkit
- PyCUDA (Python): http://mathema.tician.de/software/pycuda